ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN

GRADO EN INGENIERÍA INFORMÁTICA EN TECNOLOGÍAS DE LA INFORMACIÓN

ÁREA DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

AInotherBrickInTheWall: Aplicación de técnicas de reinforcement learning al sector de la construcción mediante agentes en Unity

D. Rivas Fernández, Alberto
TUTOR: Dª. Remeseiro López, Beatriz
COTUTORES: D. Vinci, Walter
D. Rubio Fernández, Ángel

FECHA: Julio 2023

Índice

1. Hipótesis de partida y alcance 6
1.1. INTRODUCCIÓN 6
1.2. PLANTEAMIENTO DEL PROYECTO 8
2. Objetivos concretos y relación con el estado actual 10
2.1. ESTADO DEL ARTE 10
2.1.1. Procesos de decisiones de Markov 10
2.1.2. Políticas 11
2.1.3. Funciones de valor 12
2.1.4. Funciones de optimalidad 13
2.1.5. Algoritmos empleados 14
2.1.6. Publicaciones relevantes 15
2.2. OBJETIVOS CONCRETOS 16
3. Metodología de trabajo 17
3.1. TECNOLOGÍAS EMPLEADAS 17
3.1.1. Unity 17
3.1.2. ML-Agents 18
3.1.3. Python 3 18
3.1.4. TensorBoard 18
3.1.5. C# 19
3.1.6. Visual Studio 19
3.1.7. Jupyter Notebooks y servicios en la nube 19
3.1.8. Git y Gitlab 20
3.2. DESARROLLO DEL PROYECTO 20
3.2.1. Investigación inicial 20
3.2.2. Objetivos incrementales 21
3.2.3. Ciclo de trabajo 21
4. Trabajo realizado y resultados obtenidos 23
4.1. MODELOS E HIPERPARÁMETROS 23
4.1.1. Arquitectura de la red neuronal 23
4.1.2. Hiperparámetros de entrenamiento 24
4.1.3. Señales de recompensa 26
4.2. ENTORNO CONTINUO 3D 26
4.2.1. Primer entorno 26

4.2.2. PushBrick 30
4.2.3. 2Bricks 32
4.3. ENTORNO DISCRETO 3D 34
4.3.1. Discrete 35
4.3.2. Discrete15 38
4.3.3. DiscreteSmall 39
4.3.4. DiscreteV2 40
4.3.5. dV2-5B 42
4.3.6. DiscreteV3 44
4.3.7. DiscreteV4 47
4.3.8. DiscreteV5 51
4.4. ENTORNO DISCRETO 2D 56
4.4.1. Versiones del entorno 57
4.5. SEPARACIÓN DE ENTORNOS 66
5. Conclusiones y trabajos futuros 67
5.1. CONCLUSIONES 67
5.2. TRABAJO FUTURO 68
6. Bibliografía 69

Índice de figuras

Figura 1.1: Relación entre inteligencia artificial, aprendizaje automático y aprendizaje profundo y su aparición temporal [2]. 6
Figura 1.2: Principales paradigmas del aprendizaje automático. 7
Figura 2.1: Relación entre agente y entorno en un PDM [4] 10
Figura 2.2: Árbol de estados y decisiones [4]. 13
Figura 4.1: Estructura de una red neuronal [34] 24
Figura 4.2: Vista general del primer entorno 27
Figura 4.3: Zona objetivo y su collider 27
Figura 4.4: Orientación del agente. 28
Figura 4.5: Distancia de agarrado. El bloque de la izquierda está demasiado lejos, mientras que el de la derecha está dentro del rango. 29
Figura 4.6: Sensores del agente en el entorno PushBrick. 31
Figura 4.7: Recompensa acumulada en PushBrick. 32
Figura 4.8: Agente con bloque desplazándose hacia el objetivo. 33
Figura 4.9: Recompensa acumulada en un entrenamiento del entorno 2Bricks. 33
Figura 4.10: Primer entorno discreto. 35
Figura 4.11: Sensores actualizados. 36
Figura 4.12: Recompensa acumulada en entorno Discrete. 37
Figura 4.13: Distribución de la recompensa acumulada del mejor entrenamiento del entorno Discrete. 37
Figura 4.14: Recompensa acumulada en los entrenamientos de Discrete15. 38
Figura 4.15: Mejor entrenamiento de Discrete15. 38
Figura 4.16: Estado clave para la construcción en vertical. 39
Figura 4.17: Recompensa acumulada en los entrenamientos de DiscreteSmall. 40
Figura 4.18: Sensores del agente en DiscreteV2. Los nuevos sensores se representan en color azul. 41
Figura 4.19: Recompensa acumulada en los entrenamientos de DiscreteV2. 42
Figura 4.20: Mejor entrenamiento de DiscreteV2. 42
Figura 4.21: Recompensa acumulada en los entrenamientos de dV2-5B. 42
Figura 4.22: Pasos por muro completo en dV2-5B. 43
Figura 4.23: Distribución de recompensa acumulada en dV2-5B. 43
Figura 4.24: Zona objetivo con trigger con niveles. 45
Figura 4.25: Recompensa acumulada en algunos entrenamientos de DiscreteV3. 45
Figura 4.26: Pasos por nivel en entrenamientos de DiscreteV3. 46
Figura 4.27: Sensores funcionando incorrectamente. 47
Figura 4.28: Bloques colocados según el aparejo de soga. 48
Figura 4.29: Movimiento lateral de un bloque. 48
Figura 4.30: Raycasts lanzados por el bloque A a los bloques B y C (raycast antiguo en rojo y raycasts nuevos en verde) 49
Figura 4.31: Distribución de recompensas en un entrenamiento de DiscreteV4 50
Figura 4.32: Agente bloqueado. 50
Figura 4.33: Margen entre el tablero y las paredes 51
Figura 4.34: Movimiento en distancias de media unidad. 52

Figura 4.35: Pasos por muro completado. Dos stacked vectors (azul) y sensores originales (rojo). 54
Figura 4.36: Recompensa acumulada. Dos stacked vectors (azul) y sensores originales (rojo). 54
Figura 4.37: Registro del final del entrenamiento. Pueden verse los segundos transcurridos en la primera línea. 55
Figura 4.38: Pasos por muro completo en el entrenamiento largo. 55
Figura 4.39: Distribución de recompensas en el entrenamiento largo. 55
Figura 4.40: Tileset. 57
Figura 4.41: Primer entorno bidimensional. 58
Figura 4.42: Segundo entorno bidimensional. 58
Figura 4.43: Primeros dos pasos del ejemplo. 59
Figura 4.44: Pasos tercero y cuarto del ejemplo. 60
Figura 4.45: Pasos quinto y sexto del ejemplo. 60
Figura 4.46: Primer entrenamiento. Entorno 3x3 con dos bloques. 61
Figura 4.47: Entrenamientos de un entorno 5x5 con tres bloques. 61
Figura 4.48: Entrenamiento del entorno 6x6 con 3 bloques. 62
Figura 4.49: Recompensa acumulada. Entorno 5x5 con dos niveles y seis bloques. 63
Figura 4.50: Fracción de episodios que completan el primer nivel en orden. 63
Figura 4.51: Pasos necesarios para colocar el sexto y último bloque. 64
Figura 4.52: Distribución de recompensas en el primer entrenamiento del entorno 6x6. 64
Figura 4.53: Aprendizaje de la construcción de dos niveles en orden. 65
Figura 4.54: Últimos entrenamientos del entorno 6x6 con 8 bloques. 65

Índice de ecuaciones

Ecuación (2.1) 11
Ecuación (2.2) 11
Ecuación (2.4) 12
Ecuación (2.5) 13
Ecuación (2.5) 13
Ecuación (2.6) 13
Ecuación (2.7) 14
Ecuación (2.8) 14
Ecuación (2.9) 14
Ecuación (2.10) 14

1. Hipótesis de partida y alcance

1.1. INTRODUCCIÓN

La inteligencia artificial (IA) es un campo de la informática que ha cobrado gran relevancia en las últimas décadas, con numerosos avances significativos que han revolucionado nuestras vistas. En la actualidad, su popularidad se ha visto propulsada por la aparición de ChatGPT, un *chatbot* desarrollado por OpenAI. [1]

La IA es un ámbito muy amplio que contiene muchas ramas, entre las que actualmente destacan el aprendizaje automático y el aprendizaje profundo – habitualmente conocidos por los términos en inglés *machine learning* y *deep learning*, respectivamente (ver *Figura 1.1*).

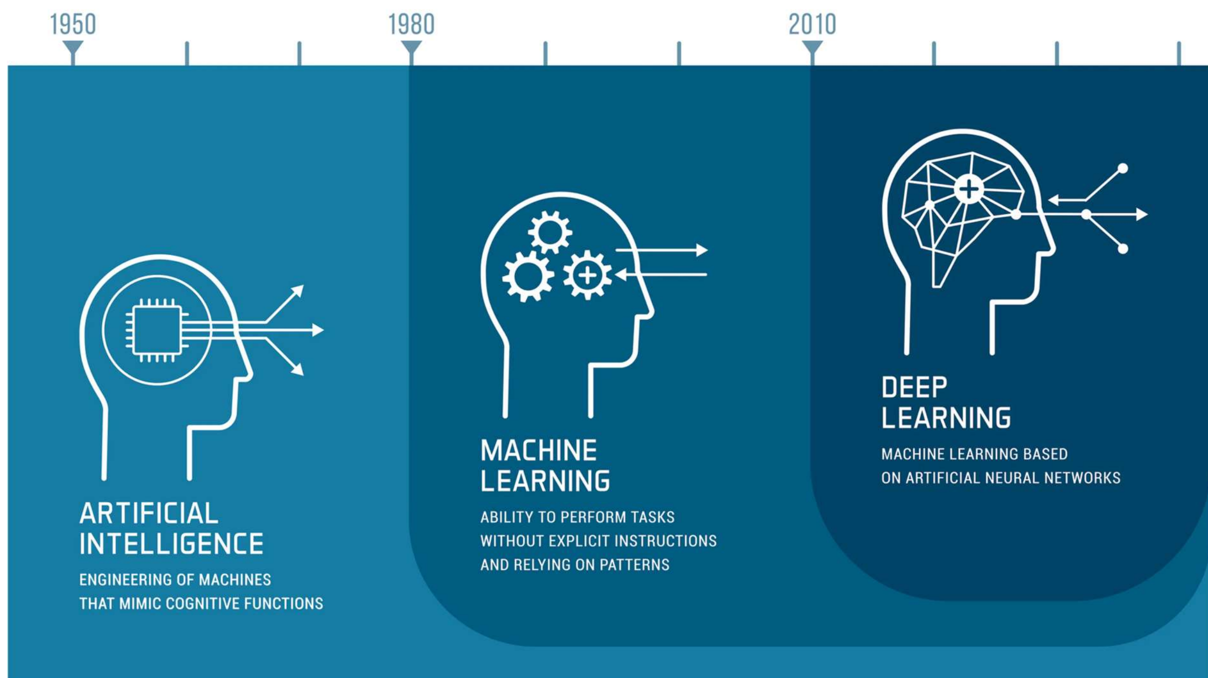

Figura 1.1: Relación entre inteligencia artificial, aprendizaje automático y aprendizaje profundo y su aparición temporal [2].

El aprendizaje automático es la rama de la IA que estudia el uso de algoritmos para extraer información relevante o identificar patrones a partir de conjuntos de datos complejos, permitiendo la solución de problemas sin intervención humana. Por su parte, el aprendizaje profundo es un subcampo del aprendizaje automático que emplea redes neuronales profundas. Estas redes neuronales están formadas por nodos – o neuronas – organizados en capas. El término profundo se refiere a las redes formadas por más de tres capas. [3]

El aprendizaje automático es un campo extenso dentro del que existen varios paradigmas. En la actualidad, los principales son el aprendizaje supervisado y el aprendizaje no supervisado.

El aprendizaje supervisado es la rama del aprendizaje automático en la que más se investiga actualmente. Consiste en entrenar un modelo utilizando un conjunto de datos previamente etiquetados. Es decir, cada muestra del conjunto de entrenamiento consta de una serie de características o atributos, y una etiqueta o valor de salida. El objetivo es que el modelo aprenda a relacionar las características de entrada con las etiquetas de salida conocidas, pudiendo así realizar predicciones precisas en datos no vistos durante la fase de entrenamiento.

Al contrario que en el aprendizaje supervisado, en el aprendizaje no supervisado no se proporcionan etiquetas o valores de salida durante el proceso de entrenamiento. En su lugar, el modelo se encarga de encontrar patrones o estructuras ocultas en los datos de entrenamiento. Este tipo de aprendizaje se utiliza para encontrar relaciones o distribuciones en los datos, proporcionado así una información muy valiosa para comprender datos complejos y tomar decisiones. Se trata, por tanto, de una herramienta muy potente para analizar datos complejos de los que no se tiene conocimiento *a priori*.

Recientemente, ha ganado relevancia un tercer paradigma que recibe el nombre de aprendizaje por refuerzo (ver *Figura 1.2*).

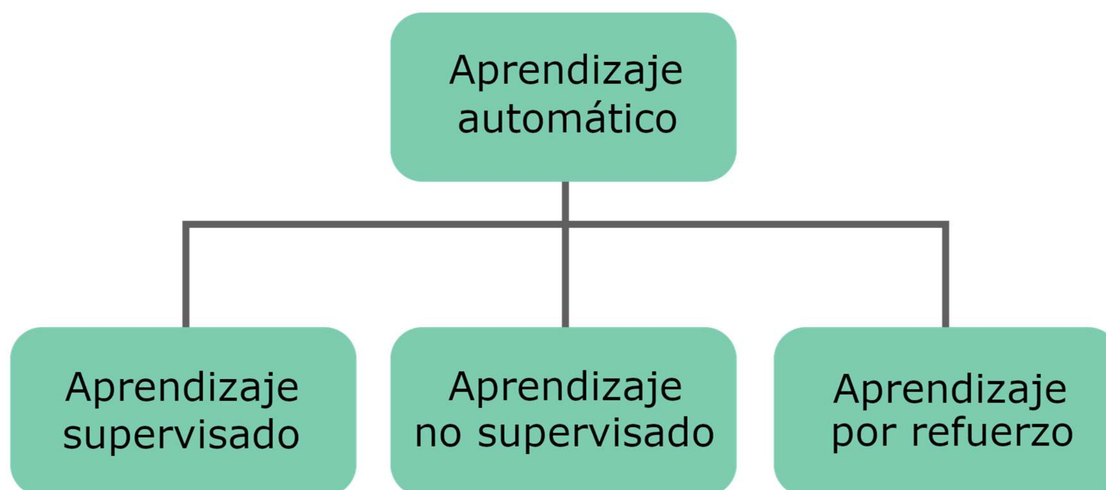

Figura 1.2: Principales paradigmas del aprendizaje automático.

El aprendizaje por refuerzo, al que es común referirse por su nombre en inglés — *reinforcement learning* (RL) — busca aprender qué hacer en diferentes situaciones con el objetivo de maximizar una señal de recompensa. El agente — el elemento que aprende — no recibe indicaciones de qué debe hacer en cada situación, sino que debe descubrir cuáles son las acciones que le proporcionan mayor recompensa mediante prueba y error. En una situación concreta, cada acción no solo devuelve una recompensa diferente, sino que también

puede modificar el entorno, llevando a una nueva situación con recompensas diferentes. Esta propiedad significa que para obtener la mayor recompensa posible a largo plazo no basta con maximizar la recompensa en cada paso. [4]

El aprendizaje por refuerzo es un campo estudiado en animales por psicólogos del comportamiento desde los años 1930. En los años 1950 se comenzaron a crear modelos computacionales basados en las teorías sobre estos comportamientos animales. Durante las décadas de 1950 y 1960 la figura que más destaca en el campo es Richard Bellman. [5]

Aunque el concepto de aprender del entorno había sido explorado durante los orígenes de la inteligencia artificial, las investigaciones se centraron rápidamente en otros ámbitos, como el aprendizaje supervisado o el reconocimiento de patrones. No fue hasta los años 1980 cuando nuevos investigadores como Richard Sutton volvieron a interesarse por el RL. Desde entonces, el volumen de interés e investigación en RL no ha parado de crecer y se ha convertido en uno de los campos más activos dentro de la inteligencia artificial. [5]

No obstante, el propio Sutton admite que, aunque nuestro conocimiento haya aumentado notablemente, el problema general del aprendizaje por interacción para conseguir objetivos concretos aún está lejos de ser resuelto. [4]

1.2. PLANTEAMIENTO DEL PROYECTO

En los últimos años, el aprendizaje por refuerzo ha demostrado ser una herramienta con un gran potencial de resolución de problemas, teniendo especial éxito su aplicación en el ámbito de los juegos. Por ejemplo, en el caso del ajedrez, un juego con multitud de expertos que han desarrollado y depurado técnicas a lo largo de cientos de años, AlphaZero aprendió rápidamente las jugadas consideradas óptimas en la actualidad. Y sin observar ninguna partida, únicamente jugando contra sí mismo. [6]

El objetivo principal de este Trabajo Fin de Grado, propuesto por la empresa HP SCDS en el ámbito del Observatorio Tecnológico HP de la Universidad de Oviedo, es aplicar el aprendizaje por refuerzo al ámbito de la construcción. En particular, se pretende entrenar a un agente para que sea capaz de erigir un muro. Para que la red neuronal asociada al agente se entrene correctamente, debe definirse un sistema de recompensas que premie la construcción adecuada y castigue los fracasos.

Desde el punto de vista del agente, habrá un objetivo final que se premiará con una recompensa: completar el muro. También habrá objetivos parciales, como por ejemplo colocar un ladrillo correctamente, por los que el agente recibirá recompensas menores, haciendo más probable que complete el objetivo final. Haciendo un paralelismo con el juego del ajedrez, el objetivo final será realizar un jaque mate y los objetivos parciales podrían ser eliminar una pieza del oponente o controlar una casilla del tablero. De igual manera, puede haber situaciones desfavorables que deben aportar recompensas negativas, como que el muro se caiga o perder una pieza del ajedrez.

Por último, cabe destacar que el planteamiento de este proyecto impone el uso de ciertas tecnologías y componentes software. En concreto, es necesario que el entorno donde se construye el muro esté desarrollado en Unity. Además, se requiere el uso de la herramienta ML-Agents para integrar agentes de IA en dicho entorno.

2. Objetivos concretos y relación con el estado actual

2.1. ESTADO DEL ARTE

El aprendizaje por refuerzo es un campo en el que se han desarrollado multitud de algoritmos y técnicas para resolver problemas que pueden llegar a ser muy complejos. No obstante, todos pueden ser definidos como procesos de decisiones de Markov con un mismo objetivo: la obtención de una política óptima. [4]

2.1.1. Procesos de decisiones de Markov

Los procesos de decisiones de Markov (PDM) son una manera de formalizar una secuencia de toma de decisiones. Cada decisión, a la que se denomina acción, retorna una recompensa y afecta a los estados siguientes.

En un PDM hay dos elementos que interactúan (ver *Figura 2.1*): el agente, que toma las decisiones y aprende; y el entorno, que es todo lo demás. El entorno responde a las acciones del agente con recompensas (valores numéricos) y se ve afectado por ellas, dando lugar a estados nuevos. El agente recibe esta información, y el ciclo se repite.

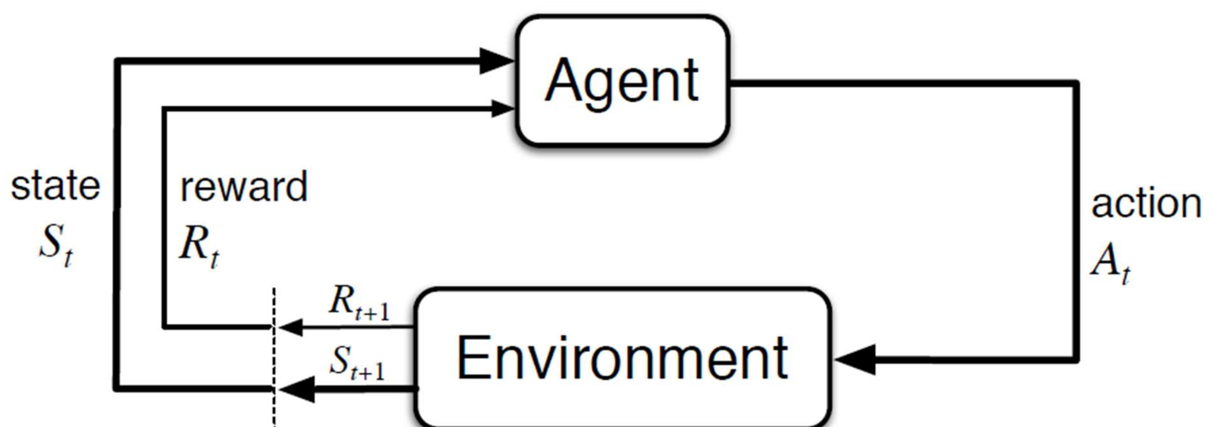

Figura 2.1: Relación entre agente y entorno en un PDM [4]

Esta primera definición del problema requiere algunas aclaraciones:

- Tiempo (t): es una variable discreta que representa los momentos en los que el agente recibe información del entorno y toma una decisión. Su notación es mediante números naturales: $t \in \{0, 1, 2, 3, \ldots\}$.
- Estado (s): pertenece al conjunto de estados posibles de cada problema, $s \in S$. Dentro de este conjunto puede haber estados terminales. Si se alcanza un estado terminal, el proceso de toma de decisiones – denominado episodio – termina.
- Acción (a): pertenece al conjunto de acciones posibles para el estado *s*, $a \in A(s)$.
- Recompensa (r): pertenece a un subconjunto de números reales predefinido en cada problema. En un ejemplo muy sencillo, el conjunto de recompensas podría ser $R = \{-1, 1\}$. Todas las acciones consideradas negativas tendrían el valor $r = -1$ y todas las positivas, $r = 1$.

Por último, cabe mencionar que la relación entre el agente y el entorno está definida por una función de distribución:

$$p(s', r \mid s, a) \doteq Prob(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a) \tag{2.1}$$

Dado el estado y la acción actuales, esta función indica la probabilidad de obtener una recompensa y un nuevo estado. El hecho de que el siguiente estado y recompensa solo dependan del estado y acción actuales, y no de la sucesión de estados y acciones pasadas, es la propiedad de Markov. [4]

2.1.2. Políticas

El comportamiento del agente está definido por una política: $\pi(a|s)$. Una política define, dado un estado, la probabilidad de seleccionar cada acción. Una buena política es aquella que, de media a lo largo de muchos episodios, acumula un retorno grande. El retorno (G_t) es la suma de recompensas obtenidas a partir de un momento t hasta que se alcanza un estado terminal en el momento T.

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{2.2}$$

Esta definición del retorno incluye un factor de descuento ($0 \leq \gamma \leq 1$) para dar un mayor peso a recompensas cercanas. Cuanto menor sea este factor, más se priorizan las recompensas a corto plazo.

Una política relaciona estados $s \in S$ y probabilidades de realizar cada acción $a \in A(s)$. Un agente que sigue una política π y se encuentra en un estado s elige aleatoriamente la acción a realizar de la distribución de probabilidad dada por $\pi(a|s)$.

La solución al problema de un PDM es una política que, en cada estado, seleccione la acción con mayor retorno. Estas políticas se denominan políticas óptimas. [4]

2.1.3. Funciones de valor

El retorno esperado de un estado bajo una política se expresa mediante la función de valor de estado.

$$v_\pi(s) \doteq \mathbb{E}[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \mid S_t = s\right], \forall\, s \in S \qquad (2.3)$$

donde $\mathbb{E}$ indica el valor esperado de una variable aleatoria y $\mathbb{E}_\pi$ el valor esperado cuando el agente sigue una política π.

De igual forma, la función de valor de acción se define como el retorno esperado de una acción en un estado dado bajo una política:

$$q_\pi(s,a) \doteq \mathbb{E}[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \mid S_t = s, A_t = a\right] \qquad (2.3)$$

Estas dos funciones pueden estimarse de forma empírica, utilizando un agente que siga la política π para explorar el entorno. Cada vez que el agente completa un episodio, se obtiene un valor para cada estado y acción por los que ha pasado. Cuando el número de episodios tiende a infinito, se tienen suficientes datos de todos los estados y acciones para que el promedio del retorno de cada uno converja al valor real.

Cuando el número de estados del PDM es muy elevado, es imposible obtener información de cada estado. En esos casos, se emplean técnicas para aproximar las funciones de valor, habitualmente transformándolas en funciones parametrizables.

Una propiedad importante de las funciones de valor es su recursividad. Esta propiedad permite relacionar formalmente el valor de un estado o acción y sus sucesores. Estas fórmulas, obtenidas a partir de las funciones de valor, son las denominadas ecuaciones de Bellman. Muchos métodos que calculan o estiman las funciones de valor se basan en ellas. La Ecuación 2.5 muestra la ecuación de Bellman para un estado s. [4]

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_t \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\big[r + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']\big] \\
&= \textstyle\sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]\,,\ \forall\, s \in S
\end{aligned}
\tag{2.4}
$$

La *Figura 2.2* ilustra un árbol de estados y decisiones en el que *s* es el estado actual, *a* representa las posibles acciones bajo una política π y *s'* los posibles sucesores. Para calcular el valor del estado actual, se suma el valor de cada estado sucesor de forma ponderada, teniendo en cuenta la probabilidad *p* de llegar al mismo (ver *(2.1)*). También incluye un factor de descuento γ para priorizar en mayor o menor medida a las recompensas a corto plazo.

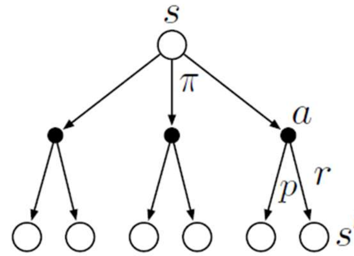

Figura 2.2: Árbol de estados y decisiones [4].

2.1.4. Funciones de optimalidad

Las políticas pueden definirse mediante sus funciones de valor. Por tanto, una política óptima π_* se define mediante las funciones de valor óptimas. En dicha política, la función de valor de estado devuelve, para cada estado *s*, un retorno mayor o igual al de cualquier otra política en dicho estado:

$$v_{\pi_*}(s) \geq v_\pi(s), \qquad \forall\, s \in S \tag{2.5}$$

La función de valor de estado de la política óptima se denomina función de valor de estado óptima y se define de la siguiente manera:

$$v_*(s) \doteq \max_\pi v_\pi(s)\,,\ \forall\, s \in S \tag{2.6}$$

Siguiendo un desarrollo equivalente para la función de valor de acción:

$$q_{\pi_*}(s,a) \geq q_\pi(s,a), \qquad \forall\, s \in S\ y\ a \in A, \tag{2.7}$$

se llega a la definición de la función de valor de acción óptima:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a)\,,\ \forall\, s \in S\ y\ a \in A \tag{2.8}$$

A partir de estas funciones se obtiene, de la forma expuesta anteriormente, las ecuaciones de optimalidad de Bellman. Para un PDM finito – es decir, con un número finito de estados *n* y acciones *m* – se obtiene un sistema de ecuaciones formado por *n* ecuaciones de estado y *n·m* ecuaciones de acción. La única solución del sistema es la política óptima.

La obtención y resolución de este sistema de ecuaciones no es, en general, una solución viable. En primer lugar, requiere un conocimiento completo del entorno y sus dinámicas, cuando en la práctica la información de la que se dispone suelen ser observaciones parciales. En segundo lugar, la capacidad computacional requerida suele ser inalcanzable. Por último, el entorno debe cumplir la propiedad de Markov. Como consecuencia, los métodos empleados en aprendizaje por refuerzo calculan soluciones aproximadas y, en la actualidad, la herramienta que se emplea para aplicar dichos métodos son las redes neuronales. [4]

2.1.5. Algoritmos empleados

Existen varias clases de algoritmos que calculan soluciones aproximadas de una política óptima, como los métodos de Monte Carlo, *Temporal Difference Learning*, y los métodos de gradiente de política. En este proyecto se emplea un algoritmo de la familia PPO (Proximal Policy Optimization), desarrollada por OpenAI en 2017. PPO pertenece a la clase de algoritmos de gradiente de política. [7] [8]

El principio de funcionamiento de los algoritmos de gradiente de política es aprender una política parametrizada:

$$\pi(a|s,\theta) = Prob\{A_t = a | S_t = s, \theta_t = \theta\} \tag{2.9}$$

La estimación del parámetro θ se realiza mediante la técnica de gradiente ascendente, buscando maximizar una métrica de rendimiento $J(\theta)$:

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)} \tag{2.10}$$

donde $\widehat{\nabla J(\theta_t)} \in \mathbb{R}^d$ es una estimación estocástica.

PPO es un algoritmo *on-policy*. Esto quiere decir que la política que optimiza a lo largo del entrenamiento es la misma que el agente emplea para explorar los diferentes estados. Este planteamiento presenta el posible problema de quedarse atascado en valores óptimos locales, en vez de explorar correctamente todos los estados posibles y obtener la política óptima real. La solución es utilizar políticas blandas, que son las que siempre mantienen una probabilidad positiva para todas las acciones en todos los estados. Poco a poco estas probabilidades deben cambiar, acercándose a la política óptima, que es determinista – es decir, en cada estado siempre selecciona la misma acción.

La otra alternativa que otros métodos utilizan se denomina *off-policy*, una solución más directa al problema de exploración versus explotación. Consiste en emplear dos políticas: una de la que se aprende y se intenta optimizar, y otra que prioriza más la exploración frente a la explotación de las mayores recompensas ya conocidas. [4]

2.1.6. Publicaciones relevantes

- Algoritmo MuZero [9]. Los resultados obtenidos supusieron un nuevo estado del arte en 2019. El algoritmo más avanzado hasta entonces era AlphaZero [10], que logró dominar juegos de mesa como el ajedrez, el go y el shogi, con el previo conocimiento de sus reglas. MuZero obtuvo los mismos resultados en esos juegos, pero sin requerir el conocimiento de las normas de antemano. Además, también logra dominar una serie de videojuegos Atari que se emplean como *benchmark* en el campo del RL [11].
- Presentación de la familia de algoritmos *Proximal Policy Optimization*, desarrollados por científicos de OpenAI [12]. Estos nuevos métodos supusieron mejoras sobre los métodos de gradiente de política existentes, y fueron probados en simulaciones de robots y en videojuegos.

Además de los estudios previamente mencionados, se han encontrado en la literatura reciente otros trabajos de investigación centrados en técnicas de aprendizaje por refuerzo (RL) en entornos Unity. A continuación, se describen brevemente dos de ellos, a modo de ejemplo.

- *Discrete Deep Reinforcement Learning for Mapless Navigation*. En este estudio se investigan algoritmos con espacios de estados discretos aplicados a la navegación sin conocimiento previo del entorno. Se consiguen resultados similares a los obtenidos con alternativas continuas, pero con tiempos de entrenamiento reducido. [13]
- *Reinforcement Learning From Hierarchical Critics*. Este estudio emplea un método de RL denominado *actor-critic* para entrenar agentes en diferentes tareas de naturaleza competitiva. Este método logra mejores resultados que el algoritmo de referencia, PPO. [14]

2.2. OBJETIVOS CONCRETOS

El objetivo inicial de este trabajo es el desarrollo de un entorno en el que un agente entrenado con RL construya un muro. Esta meta, debido a su complejidad, se planteó de forma incremental. Los primeros objetivos consistieron en tareas sencillas, aumentando la complejidad de la tarea que el agente debe completar poco a poco. Los objetivos, muy genéricos en un principio – la construcción de un muro se puede definir de muchas maneras diferentes –, fueron cambiando a lo largo del proyecto, adaptándose a los resultados obtenidos hasta el momento y a la aparente inviabilidad de algunas tareas en ciertos casos.

Además, se añadieron objetivos que no estaban contemplados en principio, relativos al diseño del entorno. En varias ocasiones a lo largo del proyecto, la complejidad del entorno – ya fuera por sus dinámicas o por la percepción de este por parte del agente – supuso un obstáculo al aprendizaje de las tareas deseadas.

Es importante diferenciar entre la complejidad de la tarea y la complejidad del entorno. Una tarea sencilla es, por ejemplo, colocar un ladrillo; y una más compleja es colocar una fila de ladrillos. El aumento de la complejidad de las tareas es, en general, inevitable. Un entorno complejo, en un problema planteado como un PDM, es uno que tiene más estados o acciones posibles o en el que la información que recibe el agente de los estados es más incompleta.

En estos casos, el objetivo pasaba a ser rediseñar o modificar algún elemento del entorno para disminuir la cantidad de estados o acciones, o modificar los sensores del agente para obtener una mejor representación del entorno.

Los objetivos concretos a lo largo del proyecto fueron los siguientes:

- Construcción de un entorno y un agente funcionales, que una persona pueda controlar.
- Integrar agentes inteligentes al entorno.
  - Aprendizaje: llevar un bloque hasta una zona predefinida de construcción.
  - Aprendizaje: depositar varios bloques dentro de la zona de construcción.
  - Aprendizaje: apilar bloques dentro de la zona.
- Construcción de un nuevo entorno más simple.
  - Aprendizaje: colocar los bloques en niveles completos, ocupando toda la anchura del muro.
- Construcción de un tercer entorno más simple.
  - Aprendizaje: colocar los bloques en orden, completando cada nivel antes de comenzar el siguiente.

3. Metodología de trabajo

3.1. TECNOLOGÍAS EMPLEADAS

En esta sección se describen brevemente los diferentes elementos software empleados en la realización del proyecto. El uso de algunas de estas tecnologías es un requisito inicial y constituyen el núcleo del proyecto. Otras han sido escogidas por su compatibilidad con las anteriores y por familiaridad con su uso.

3.1.1. Unity

El principal entorno de desarrollo del proyecto es Unity, un motor de videojuegos multiplataforma ampliamente utilizado hoy en día, no sólo en la industria de los videojuegos sino también en ámbitos como el cine, la arquitectura o la ingeniería. En el campo de la inteligencia artificial, es destacable su uso por DeepMind, una empresa filial de Google que saltó a la fama en 2016 cuando su programa AlphaGo derrotó al jugador de go profesional Lee Sedol. En la actualidad continúa siendo reconocida por el uso de la inteligencia artificial aplicada a juegos como el ajedrez, el shogi o el go. [15] [16]

Cada proyecto en Unity se compone de una o varias escenas. Una escena es un entorno en el que se colocan los elementos del juego o aplicación, como personajes, obstáculos, interfaces, etc. Todos estos elementos son de la misma clase: *gameObject*. Las características de cada *gameObject* están definidas por sus componentes. Existen multitud de componentes predeterminados para todo tipo de necesidades. En este proyecto, los más importantes son los que definen propiedades físicas del *gameObject*. También pueden crearse componentes a medida denominados *scripts*, necesarios para desarrollar comportamientos más complejos.

En este proyecto, Unity se utiliza para crear un entorno virtual con el que interactuará un agente al que se debe entrenar. Se busca que este entorno sea lo más realista posible. Por tanto, la característica fundamental que se debe considerar de Unity es su motor de física [17]. La simulación realista del comportamiento de objetos en un entorno tridimensional se basa en dos elementos principales:

- ***Rigibdodies.*** Un *rigidbody* es un componente que aporta cualidades físicas a un objeto, permitiendo someterlo a fuerzas, como por ejemplo masa o coeficiente de resistencia, o si le afecta o no la gravedad.
- ***Colliders.*** Un *collider* aporta a un objecto la capacidad de chocarse con otros y detectar las colisiones. Sin este componente, los diferentes objetos se atravesarían sin interactuar.

Por último, cabe mencionar que los *raycasts* son otro elemento fundamental utilizado para obtener información. Se trata de sensores lineales, con un punto de origen y otro de destino, que detectan a los objetos (en particular, a su *collider*) que se interponen en su recorrido.

3.1.2. ML-Agents

ML-Agents [18] es una herramienta de código abierto cuyo objetivo es crear comportamientos inteligentes, habitualmente empleados para jugadores virtuales o personajes no controlables en los juegos. Para lograr este fin, se entrenan agentes inteligentes mediante el uso de *reinforcement learning* o *imitation learning* – una subcategoría del aprendizaje automático basado en observar e imitar el comportamiento de expertos. El método de *imitation learning* disponible en ML-Agents se denomina *Behavioral Cloning* y emplea aprendizaje supervisado.

ML-Agents consta de dos partes: una en Python 3 y otra que se integra en el editor de Unity. El paquete de Unity permite crear agentes dentro del entorno virtual. La parte Python contiene los algoritmos de aprendizaje y una interfaz para interactuar con la escena de Unity en la que se encuentra el agente.

3.1.3. Python 3

Python [19] es uno de los lenguajes de programación más utilizados mundialmente. Es muy versátil y cuenta con infinidad de bibliotecas para ampliar sus funcionalidades, habitualmente desarrolladas por terceros. Destaca su uso en la investigación científica, con bibliotecas especializadas para multitud de áreas, como la astronomía [20], la biología [21], el análisis de Big Data [2 o el aprendizaje automático [23].

La mayoría del software de aprendizaje automático en Python utiliza como base una de dos bibliotecas: PyTorch [24] o TensorFlow [25]. Ambas bibliotecas proporcionan herramientas parecidas para crear redes neuronales, entrenar los modelos, monitorizar el entrenamiento, etc. La biblioteca de aprendizaje automático empleada por ML-Agents es PyTorch.

La ejecución de ML-Agents en Python se ha realizado en un entorno virtual creado en Python 3.7.9. En este entorno se instalan únicamente los paquetes necesarios para el proyecto. De esta manera, se evitan posibles interferencias de otras instalaciones en Python y se tiene un entorno fácilmente replicable en caso de utilizar un ordenador diferente.

3.1.4. TensorBoard

La herramienta que proporciona ML-Agents para monitorizar el entrenamiento es TensorBoard, que se ejecuta en el mismo entorno Python. TensorBoard es una herramienta de código abierto desarrollada por TensorFlow, organización desarrolladora de la biblioteca homónima, pero que puede ser utilizada con cualquier infraestructura de *deep learning*. [26]

La monitorización del entrenamiento es una herramienta muy potente. Es la única fuente de información sobre el rendimiento del entrenamiento aparte del resultado final, el modelo

entrenado. Estos datos sirven para comparar entrenamientos con diferentes hiperparámetros, conocer si el agente está realizando determinadas tareas y en cómo de eficiente es en ellas. TensorBoard informa en tiempo real, permitiendo abortar entrenamientos que no logran los objetivos, sin tener que esperar varias horas a que se complete el proceso.

Por defecto, TensorBoard muestra mediciones y gráficos de ciertas métricas del entrenamiento, organizadas en las categorías “Entorno”, “Pérdidas” y “Política”. También permite añadir nuevas estadísticas y categorías creadas a medida. Es posible acceder a las diferentes métricas y gráficas que ofrece la herramienta conectándose a un puerto local (6006 por defecto) con un navegador web. [27]

3.1.5. C#

Dentro de un proyecto Unity, el único lenguaje de programación soportado de forma nativa es C#. Esto significa que utilizar otro lenguaje para escribir *scripts* haría necesario compilarlos en un IDE externo y, posteriormente, integrarlos al proyecto. Además de su compatibilidad con Unity, C# es un lenguaje orientado a objetos que cumple todas las necesidades del proyecto, así que se ha empleado para escribir el código de los *scripts*.

Los *scripts* habituales definen una clase que hereda de *MonoBehaviour*, una clase innata de Unity que incluye ciertos métodos básicos [28]. En el caso del *script* de los agentes, se hereda de la clase *Agent*, aportada por el paquete ML-Agents. Esta clase incluye varios métodos específicos además de los básicos de *MonoBehaviour*.

3.1.6. Visual Studio

Para editar el código de los *scripts* C# se ha empleado Visual Studio Enterprise 2022. Esta versión de Visual Studio es un entorno de desarrollo integrado (IDE, por sus siglas en inglés) con multitud de prestaciones, aunque su función en el proyecto se ha limitado a ser un editor de código para Unity.

Visual Studio también se ha empleado para gestionar el repositorio Git.

3.1.7. Jupyter Notebooks y servicios en la nube

Como método para separar el entorno de desarrollo (Unity) y el de entrenamiento (Python) en máquinas distintas se han empleado cuadernos Jupyter; una tecnología de código abierto que soporta entornos de ejecución en varios lenguajes — Python en este caso — que se encuentra disponible en varios servicios web.

Esta separación de entornos no es necesaria, pero permite el aprovechamiento de infraestructuras hardware basadas en servicios en la nube, mucho más potentes y especializadas que las disponibles en un ordenador personal. Esto resulta especialmente

beneficioso en escenarios que requieren un uso intensivo de tarjetas gráficas (GPUs, por sus siglas en inglés).

El servicio en la nube escogido para este fin ha sido Google Colaboratory. El principal motivo fue la facilidad que ofrece para compartir cuadernos, permitiendo la revisión y ejecución de código por parte de los tutores.

Como apoyo al uso de cuadernos Jupyter en Colaboratory, ha sido necesario el uso de un sistema de almacenamiento en la nube. El sistema elegido fue Google Drive.

3.1.8. Git y Gitlab

Git es el sistema de control de versiones distribuido más ampliamente empleado hoy en día. Es comúnmente utilizado para almacenar y coordinar el trabajo de varias personas en un proyecto, sincronizando repositorios locales en los ordenadores de los desarrolladores con un repositorio remoto. Ofrece múltiples ventajas a la hora de gestionar los repositorios, siendo notables la posibilidad de crear ramas paralelas y su posterior fusión y la compatibilidad con múltiples protocolos. [29]

Gitlab es un servicio web en el que se almacenan repositorios Git. Originalmente era un mero servidor Git, pero actualmente también incluye herramientas orientadas a monitorizar, documentar e incluso desplegar proyectos. [30]

En este proyecto, el uso de Git y Gitlab se ha limitado a la monitorización del desarrollo de los componentes software.

3.2. DESARROLLO DEL PROYECTO

El desarrollo del proyecto se ha organizado siguiendo una metodología ágil, con *sprints* de dos semanas. Tras cada *sprint*, se revisaba el progreso y se planteaban nuevos objetivos en una reunión con los tutores de la universidad y de la empresa HP SCDS.

3.2.1. Investigación inicial

Las primeras semanas estuvieron dedicadas a familiarizarme con las técnicas de aprendizaje por refuerzo y las herramientas que se emplearían en el desarrollo del proyecto. Con ese fin, los tutores me indicaron fuentes de información para iniciarme en tres áreas principales:

- **Aprendizaje por refuerzo**: En primer lugar, estudié los conceptos básicos del aprendizaje por refuerzo utilizando el libro *Reinforcement Learning: An Introduction*

[4]. Ya que nunca antes había trabajado con esta rama del aprendizaje automático, fue importante entender en qué consiste un problema de este tipo.

- **Unity**: En este caso, el material empleado fueron vídeos educativos de diferentes canales de YouTube. Como pude comprobar a lo largo del desarrollo del proyecto, esta plataforma cuenta con una gran cantidad de divulgadores especializados en Unity. En este primer período, y siendo mi primera experiencia con Unity, me resultaron de gran ayuda los vídeos para principiantes de *Game Maker's Toolkit* [31] y *Brackeys* [32].

- **ML-Agents**: Los desarrolladores de esta herramienta ofrecen proyectos de ejemplo listos para entrenar. Siguiendo los pasos indicados en su documentación [33] [34], me familiaricé con el funcionamiento básico de ambas partes de ML-Agents, tanto dentro de Unity como en lo que respecta al entrenamiento en Python.

3.2.2. Objetivos incrementales

El resto del proyecto se planteó de forma incremental, debido a la complejidad del objetivo final — la construcción de un muro. La línea de trabajo ideada en este punto contemplaba ciertos pasos y, tras completarlos, decidíamos cómo seguir avanzando en las reuniones periódicas con los tutores. Estos primeros pasos consistían en:

1. Desarrollo de un entorno tridimensional en Unity, sin integrar ML-Agents. Consta de un agente controlable y uno o varios bloques iguales (con forma de prisma rectangular). El agente debe ser capaz de interactuar con los bloques y desplazarlos de forma controlada.
2. Integración de ML-Agents. Al entorno creado anteriormente se añade una zona objetivo, a la que el agente debe llevar un bloque. Este paso incluye tanto las modificaciones necesarias en Unity como el entrenamiento satisfactorio de la tarea.
3. Aumento de la complejidad de la tarea. El primer incremento del proyecto una vez que ML-Agents se ha integrado satisfactoriamente consiste en aumentar el número de bloques que deben ser llevados al objetivo de 1 a 2.

3.2.3. Ciclo de trabajo

Dentro de cada *sprint*, se ha trabajado siguiendo un ciclo al estilo de las metodologías ágiles, con los siguientes pasos:

- Cambios en el entorno Unity o desarrollo de un nuevo entorno.

- Cambios en la configuración del entrenamiento, que está definida en un fichero de texto con extensión *.yaml* en el que se incluyen los hiperparámetros empleados.

- Entrenamiento monitorizado. Durante el entrenamiento, que puede durar de varias horas a días en los casos más complejos, se emplea TensorBoard para comprobar si el progreso es el esperado.
- Evaluación de resultados. En primer lugar, se evalúan los valores obtenidos al final del entrenamiento en TensorBoard. Si los resultados son los esperados, se importa el modelo entrenado al entorno Unity y se ejecutan varios episodios para observar su comportamiento.

4. Trabajo realizado y resultados obtenidos

Este apartado detalla el proceso de diseño del agente y el entorno, así como los entrenamientos realizados a lo largo de los ciclos de trabajo de todo el proyecto. Todos los ficheros descritos a continuación se encuentran en el siguiente repositorio de GitLab: https://gitlab.com/HP-SCDS/Observatorio/2022-2023/ainotherbrickinthewall/epi-ainotherbrickinthewall/

4.1. MODELOS E HIPERPARÁMETROS

En este apartado se explican las configuraciones de la red neuronal utilizada en los diferentes entrenamientos. Además, se describen los hiperparámetros y otras configuraciones empleadas que, en líneas generales, se mantuvieron constantes a lo largo de los diferentes entornos, a excepción de pequeños cambios como el tamaño de la red neuronal. Esto es debido a que, una vez encontrados unos valores funcionales, suelen seguir siéndolo para los entornos subsecuentes, siendo sus entrenamientos muy similares entre sí.

Debido a la limitación temporal de un Trabajo de Fin de Grado y al elevado número de entornos y configuraciones analizadas, no se ha intentado hacer un ajuste óptimo de hiperparámetros. Las pruebas con diferentes valores se han limitado a encontrar una configuración funcional para el correcto desarrollo de los entrenamientos, dedicando la mayor parte del tiempo y esfuerzo en optimizar el diseño del entorno y el agente.

La configuración de un entrenamiento se define en un fichero de texto. Las diferentes configuraciones empleadas en este proyecto se agrupan en tres categorías principales: arquitectura de la red neuronal, hiperparámetros y señales de recompensa. [35]

4.1.1. Arquitectura de la red neuronal

- *Num layers:* Define el número de capas ocultas de la red (ver *Figura 4.1*). Para problemas sencillos, menos capas logran entrenar más rápido y eficientemente. Para problemas más complicados, es necesario un mayor número de ellas. En este trabajo, se utilizaron inicialmente redes con 2 o 3 capas ocultas, pero a medida que se fueron complicando los entornos y acciones se llegaron a definir capas con hasta 4 capas ocultas. Además de las capas ocultas, las redes neuronales tienen una capa de entrada,

cuyo tamaño corresponde al del vector de observaciones; y una capa de salida, que depende del conjunto de acciones del problema a resolver.

- *Hidden units:* Define el número de nodos en cada una de las capas ocultas de la red. Para problemas sencillos, el valor debe ser bajo, mientras que los problemas que requieran combinar observaciones complejas para decidir qué acción tomar necesitan valores mayores. Inicialmente, se emplearon capas de 256 nodos, que se ampliaron a 512 en los entornos más complejos.

- *Normalize*: Permite normalizar los datos de entrada (vector de observaciones), pudiendo resultar útil en problemas continuos complejos. En las pruebas realizadas tanto en entornos tridimensionales como bidimensionales, su uso no supuso ninguna alteración en los resultados.

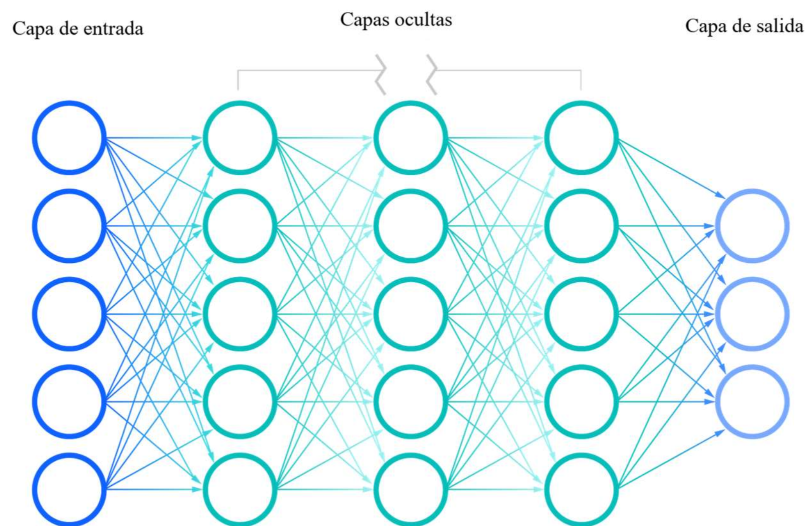

Figura 4.1: Estructura de una red neuronal [36]

4.1.2. Hiperparámetros de entrenamiento

- *Max steps*: Número total de pasos de entrenamiento, teniendo en cuenta que un paso incluye una observación del entorno y una acción del agente. En cada escena hay varios agentes iguales (40 en la mayoría de las configuraciones) funcionando en paralelo y todos contribuyen al total de pasos.

- *Buffer size*. Cantidad de experiencias – información del entorno medida en cada paso – que deben ser recogidas antes de actualizar la política. En general, cuanto mayor sea este valor, más estable es el aprendizaje. El tamaño de buffer en los primeros entrenamientos fue 2048 pero rápidamente se aumentó a 10240, valor empleado en la mayoría de los entrenamientos.

- *Num Epoch*: Número de ciclos a través del *buffer* de experiencia en el proceso de optimización mediante descenso de gradiente. Valores bajos garantizan un aprendizaje más estable pero más lento. Para emplear valores más altos del número de ciclos es necesario incrementar también el tamaño del *buffer*. Se emplearon 3 *epochs* en la mayoría de los entrenamientos, ya que las pruebas realizadas con valores mayores resultaron en aprendizajes inestables que no lograban incrementar la recompensa obtenida. En el entorno *DiscreteV5* se observó la necesidad de realizar entrenamientos más largos, por lo que se aumentó el número de *epochs* a 5, logrando mantener entrenamientos estables.

- *Batch size*. Cantidad de experiencias empleadas en cada iteración del descenso de gradiente. En cada iteración, se extrae un *batch* del buffer de experiencia. Al terminar de utilizar todos los datos del buffer de esta manera, se completa un *epoch* (y se repite el proceso tantas veces como determine el hiperparámetro *num epoch*). Como las acciones del agente son discretas, se recomienda que su valor esté entre 32 y 512. El valor utilizado es 128.

- *Learning rate*: Valor inicial del índice de la tasa de aprendizaje, que indica cuánto puede cambiar la política en cada actualización mediante descenso de gradiente. Este valor debe ser disminuido si la recompensa acumulada no aumenta a lo largo del entrenamiento.

- *Learning rate schedule*: Determina cómo cambia el valor del *learning rate* a lo largo del entrenamiento. Las opciones son:

  - *Constant:* El *learning rate* se mantiene constante durante todo el entrenamiento.

  - *Linear:* El *learning rate* se reduce linealmente hasta llegar a 0 al final del entrenamiento (definido en *max steps*). En aquellos entrenamientos en los que exista un límite de tiempo se recomienda utilizar esta opción para una mayor estabilidad, por lo que ha sido la configuración empleada – salvo en alguna prueba sin éxito.

- *Beta*: Fuerza de la regularización de la entropía, que permite garantizar que los agentes exploren de forma adecuada todas las acciones posibles. Su valor debe adaptarse para que la entropía disminuya a medida que aumenta la recompensa. Por defecto, su valor cambia linealmente a lo largo del entrenamiento.

- *Epsilon*: Indica el umbral de diferencia aceptable entre políticas nuevas y viejas durante las actualizaciones del descenso de gradiente. Afecta a la velocidad a la que puede cambiar la política durante el entrenamiento. Valores bajos conllevan actualizaciones más estables, con una mejor convergencia a la política óptima, pero ralentizan el proceso de entrenamiento. El valor empleado en la mayoría de los entrenamientos es 0,2. Cuando los entrenamientos comenzaron a necesitar más de 40 millones de pasos, se aumentó su valor hasta 0,3.

- *Lambda*: Regula en qué medida se depende de la recompensa actual (valores mayores) y de la estimación anterior (valores menores) a la hora de actualizar las

funciones de valor. Se han considerado diferentes valores en el rango recomendado [0,9, 0,95], pero no se han observado cambios relevantes en los resultados de los entrenamientos.

- *Time horizon*: Define, en número de pasos, la cantidad de experiencia que acumula cada agente antes de añadirla al *buffer*. Cuanto mayor sea, las estimaciones serán menos sesgadas pero su desviación será mayor. Es importante que este valor sea suficientemente grande para capturar todas las acciones importantes que llevan a una recompensa.

4.1.3. Señales de recompensa

- *Gamma*: es el factor de descuento aplicado a las recompensas futuras, de la misma manera que se empleaba en las *Ecuaciones 2.2*, *2.3*, *2.4* y *2.5* del apartado 2.1. Su valor por defecto es 0,99, y su rango recomendado es 0,8 – 0,995. Se recomienda disminuir si las recompensas son inmediatas, pero como las tareas en este proyecto requieren una secuencia más o menos larga de acciones, se utiliza el valor por defecto.

4.2. ENTORNO CONTINUO 3D

Para desarrollar el entorno, se plantea un único proyecto Unity en el que se crean diferentes escenas, orientadas a cumplir los objetivos incrementales asumidos a lo largo del proyecto. Cada escena se crea a partir de la anterior, modificando o añadiendo componentes. De esta manera, se puede reutilizar el código que no requiera cambios entre cada escenario.

4.2.1. Primer entorno

En primer lugar, se creó un entorno tridimensional sin integrar un agente inteligente, únicamente con componentes sencillos de Unity y código C#. Este primer entorno se ilustra en la *Figura 4.2*.

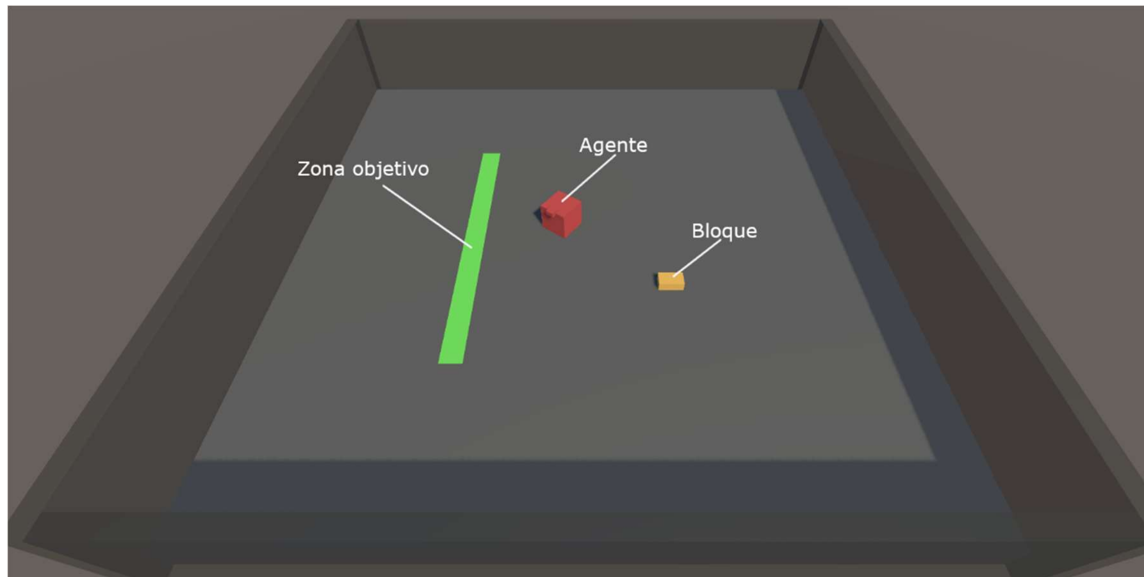

Figura 4.2: Vista general del primer entorno

Los componentes de este primer entorno son los siguientes:

- **Suelo y paredes externas.** Representan los límites del entorno. El suelo es la plataforma sobre la que se desliza el agente y se apoyan los bloques. Ofrece rozamiento para frenar el movimiento del agente y de los bloques empujados. Las paredes evitan que el agente o los bloques se caigan de la plataforma.
- **Zona objetivo.** Representada visualmente como un rectángulo verde en el suelo, hace referencia al área sobre la que se debe colocar el bloque. Tiene un *collider* atravesable — denominado *trigger* — sobre toda su superficie, que puede verse en la *Figura 4.3*.

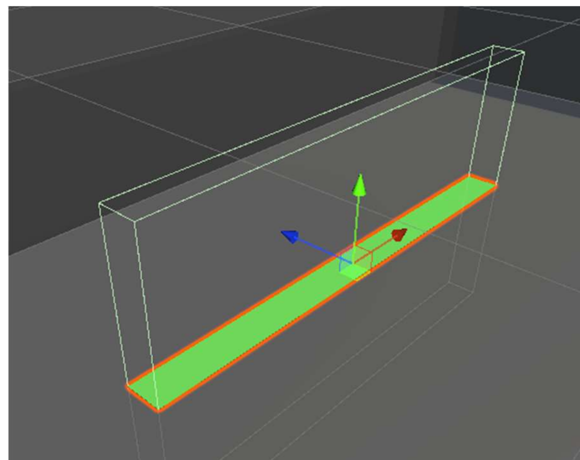

Figura 4.3: Zona objetivo y su collider.

- **Bloque.** Este objeto tiene los componentes necesarios para que su comportamiento se rija por el motor de física, un *rigidBody* y un *collider*. Así, el bloque puede chocarse contra las paredes que limitan el área y contra el agente — es decir, el agente puede empujar el bloque moviéndose contra él. Además, cuenta con dos *scripts*:
    - *GoalDetect.cs* detecta cuándo está en contacto con el *trigger* de la zona objetivo. Para ello, Unity aporta tres funciones diferentes: *OnTriggerEnter*, *OnTriggerStay* y *OnTriggerExit*, que se disparan cuando se entra en contacto, cada fotograma que se mantenga el contacto y cuando se pierde el contacto, respectivamente. Este *script* simplemente utiliza estas funciones para llamar a tres métodos que se implementarán en el agente.
    - *Pickable.*cs es el componente que marca a los bloques como objetos que el agente puede coger con su método *grabBrick*. Cuando el agente coge un bloque, este *script* se encarga de cambiar el comportamiento del bloque temporalmente, desactivando la gravedad y moviéndolo suavemente siguiendo el movimiento del agente. Igualmente, cuando el agente lo suelta, es el encargado de restaurar el comportamiento habitual.
- **Agente.** Este primer agente es un simple personaje jugable, que registra los comandos introducidos por teclado para realizar diferentes acciones:
    - Movimiento. El agente puede moverse en cuatro sentidos, mirando el área de forma cenital: arriba, abajo, derecha e izquierda. Al moverse en una dirección, también rota hasta moverse de frente (la parte frontal tiene un saliente para identificarla, tal y como se ilustra en la *Figura 4.4*).

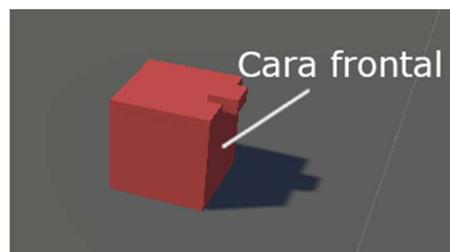

Figura 4.4: Orientación del agente.

    - Coger un bloque. El agente utiliza un *RayCast* para detectar un bloque frente a él, a una corta distancia (ver *Figura 4.5*). Si lo encuentra, lo agarra. Un bloque agarrado pasa a una posición fija respecto al agente, permitiendo moverlo de forma controlada. Si el agente ya tiene un bloque agarrado, esta acción no hace nada.

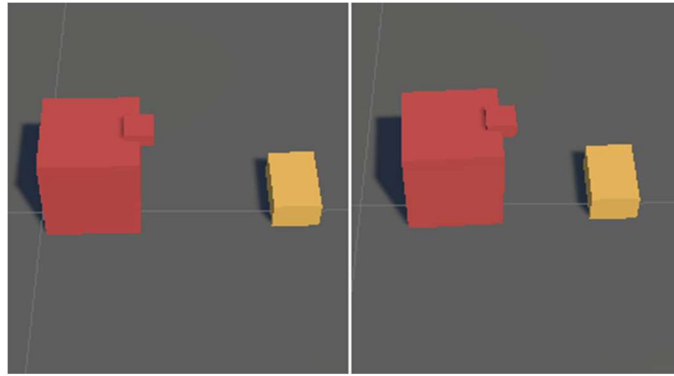

Figura 4.5: Distancia de agarrado. El bloque de la izquierda está demasiado lejos, mientras que el de la derecha está dentro del rango.

- Soltar un bloque. Se suelta el bloque actualmente agarrado. Para evitar comportamientos excesivamente complicados, el componente *Pickable.cs* del bloque se encarga de que caiga en la posición actual, poniendo a cero su velocidad y rotación. Si el agente no tiene un bloque agarrado, esta acción no hace nada.
- Rotar un bloque. Rota 90° horizontalmente el bloque agarrado. Si no hay ningún bloque agarrado, no hace nada.
- Levantar un bloque. Mueve el punto de agarre hacia arriba una pequeña distancia fija (aproximadamente la altura de un bloque). Si no hay ningún bloque agarrado o ya se ha alcanzado una altura máxima, no hace nada. El punto de agarre vuelve a su posición original tras soltar un bloque.
- Bajar un bloque. Equivalente a la acción anterior, pero bajando en punto de agarre.

El agente también se encarga de colocar en una posición y rotación aleatorias la zona objetivo al comienzo de la ejecución.

- **BrickSpawner.** Es el componente que se encarga de generar los bloques en posiciones y rotaciones aleatorias al comienzo de la ejecución, comprobando que esas posiciones no están ocupadas por la zona objetivo, el agente u otros bloques.

Este primer entorno ya es funcional, pero aún le faltan varios elementos clave para representar un problema de aprendizaje por refuerzo. Tiene un agente con la capacidad de realizar todas las acciones necesarias y un entorno con el que interactúa. A continuación, debe introducirse un sistema de recompensas, la definición de un episodio y dotar al agente de la capacidad de observar el entorno y tomar decisiones. Todo esto se introduce con ML-Agents.

4.2.2. PushBrick

Se trata del primer entorno completo con un agente inteligente, construido sobre la escena anterior. En este paso se ha añadido el paquete ML-Agents al proyecto Unity y se ha modificado el agente. La clase *BuilderAgent* ahora hereda de *Agent*, una clase proporcionada por ML-Agents. El código existente pasa a organizarse en los métodos heredados.

- El código que inicializa de forma aleatoria el entorno pasa a estar dentro del método *OnEpisodeBegin*, que se ejecuta automáticamente al comenzar cada episodio.
- Se añaden tres métodos para añadir las siguientes recompensas:
  - Cuando un bloque entra en contacto con la zona objetivo, añade una recompensa positiva pequeña, con valor 1.
  - Cuando un bloque entra por completo dentro de la zona objetivo, una recompensa positiva de valor 2 y lo marca como colocado, utilizando el sistema de etiquetas de Unity.
  - Cuando un bloque etiquetado como colocado deja de estar completamente dentro de la zona objetivo, añade una recompensa negativa de valor -2 y lo vuelve a etiquetar como bloque sin colocar.
  - Cuando un bloque deja de estar en contacto con la zona objetivo, añade una recompensa negativa pequeña, con valor -1.
- Se añade una llamada a *EndEpisode* tras lograr el objetivo. Este método termina el episodio actual, tras el que comienza el siguiente automáticamente. La llamada a *EndEpisode* se coloca tras añadir la recompensa correspondiente a colocar un bloque en la zona.
- Se añaden sensores al agente. Los sensores son componentes proporcionados por ML-Agents que permiten recoger información del entorno. Los sensores escogidos son lineales, detectando el primer *collider* con el que se encuentran. Se han colocado en dos niveles, orientados en abanico como se observa en la *Figura 4.6*.

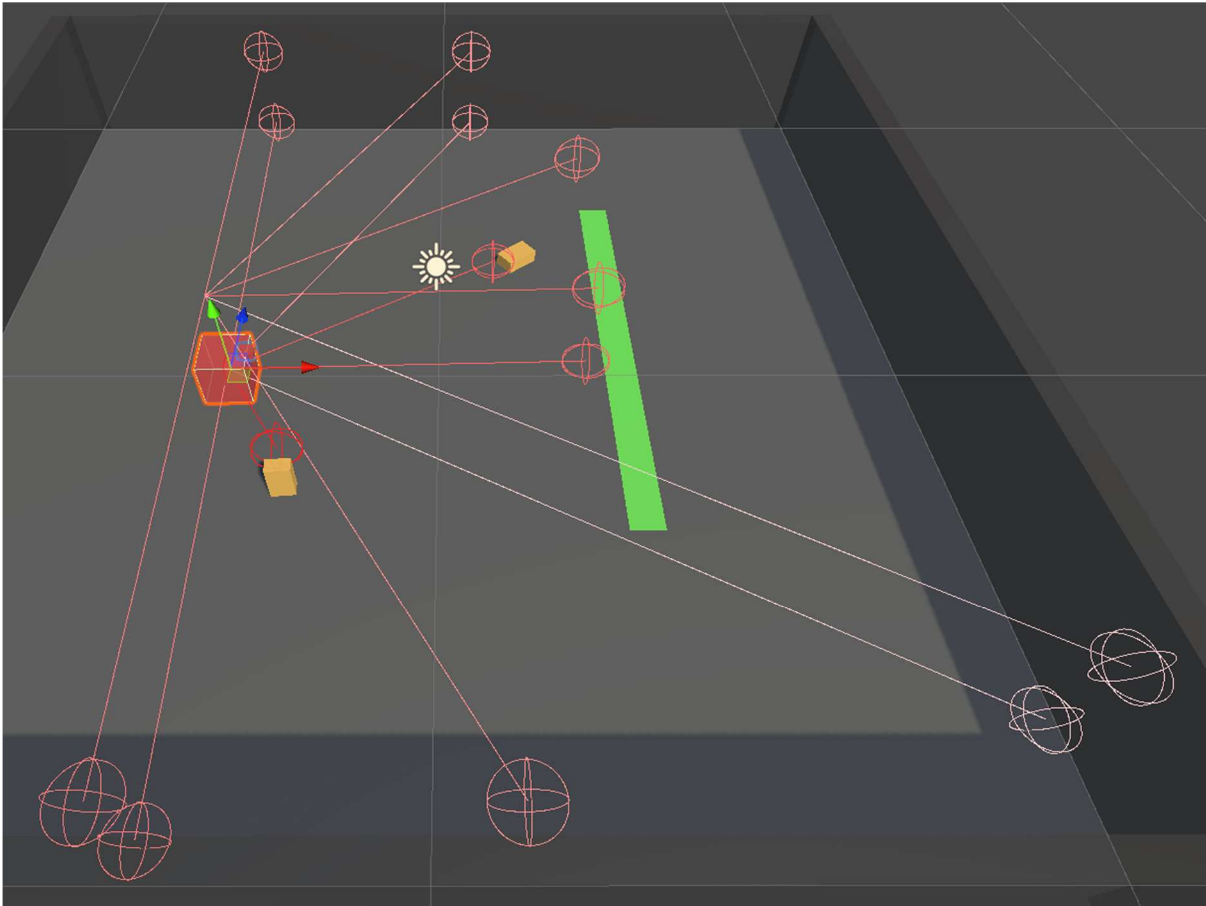

Figura 4.6: Sensores del agente en el entorno PushBrick.

- Se añade el componente *DecisionRequester*, que determina la frecuencia de la toma de decisiones. Cada decisión añade una recompensa negativa muy pequeña (valor acumulado de –1 en un máximo de 5000 pasos) para incentivar la eficiencia en las acciones elegidas. Así, los episodios que se completen en menos pasos recibirán recompensas mayores.
- No se incluyen dentro de las acciones del agente las acciones de rotar, levantar y bajar el bloque agarrado. Estas tres acciones no son necesarias para la tarea deseada – meter un bloque en la zona. Cuanto más reducido sea el conjunto de acciones, más simple es el PDM que se trata de resolver.

Entrenamiento y resultados

El primer paso del entrenamiento es definir el archivo de configuración, que contiene los hiperparámetros. Para configurar el primer entrenamiento se ha tomado como base el archivo de configuración de *PushBlock*, un entorno similar perteneciente al proyecto de ejemplo de ML-Agents. Para aprender a calibrar todos los hiperparámetros se ha consultado la documentación proporcionada en el repositorio de ML-Agents en GitHub. [35]

La métrica empleada es la recompensa acumulada (ver *Figura 4.7*). Tras varias pruebas, se obtiene un resultado satisfactorio, correspondiente a la línea morada en la gráfica. La recompensa debe aproximarse al valor máximo lo más rápido posible y luego mantenerse estable con el menor ruido posible. En este caso, la recompensa por completar la tarea con éxito es 2.

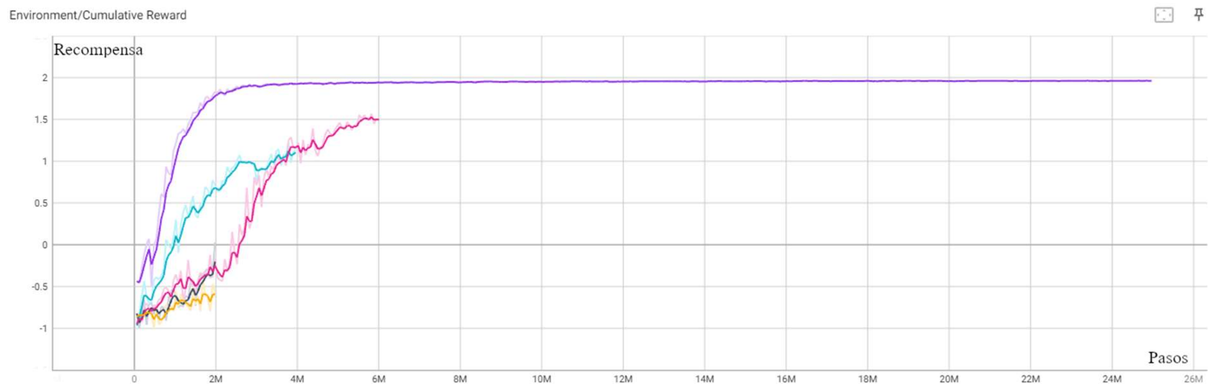

Figura 4.7: Recompensa acumulada en PushBrick.

4.2.3. 2Bricks

La primera modificación del entorno fue realizada en la misma escena. En este caso se utilizan dos bloques en vez de uno y se modifica levemente el código de *BuilderAgent.cs* para que el episodio termine cuando haya dos bloques dentro de la zona.

La complejidad de esta tarea, aunque en principio no lo parezca, es mucho mayor que la del caso anterior. En la primera versión del entorno, *PushBrick*, el agente aprendía a coger un bloque y desplazarse contra la zona objetivo (ver *Figura 4.8*). Una vez que el bloque entra por completo, el episodio terminaba de forma exitosa.

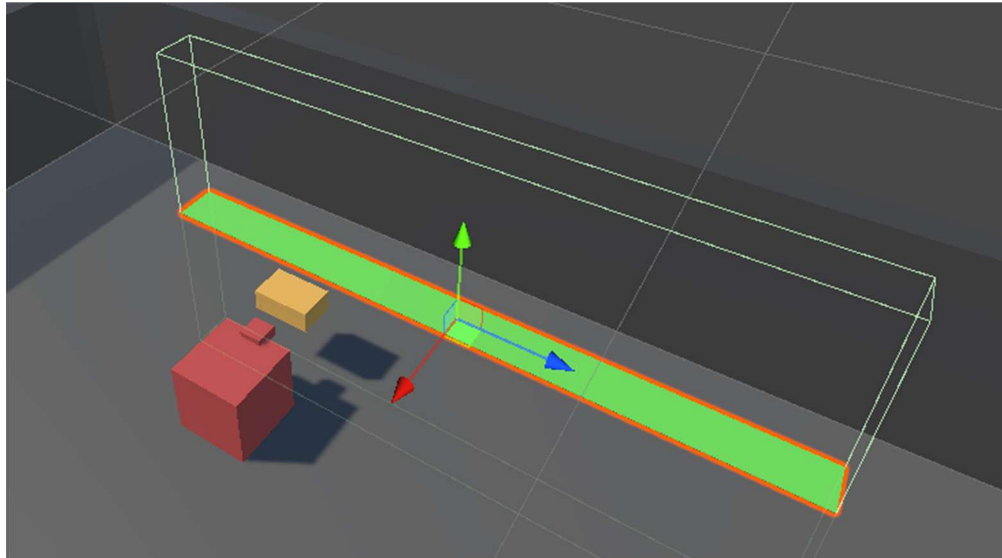

Figura 4.8: Agente con bloque desplazándose hacia el objetivo.

En esta tarea, el agente debe aprender, además, a soltar el primer bloque dentro del objetivo antes de ir a por el segundo bloque y llevarlo a la zona marcada. El incremento en complejidad hizo necesario aumentar el tamaño de las capas ocultas de la red neuronal de 256 nodos a 512.

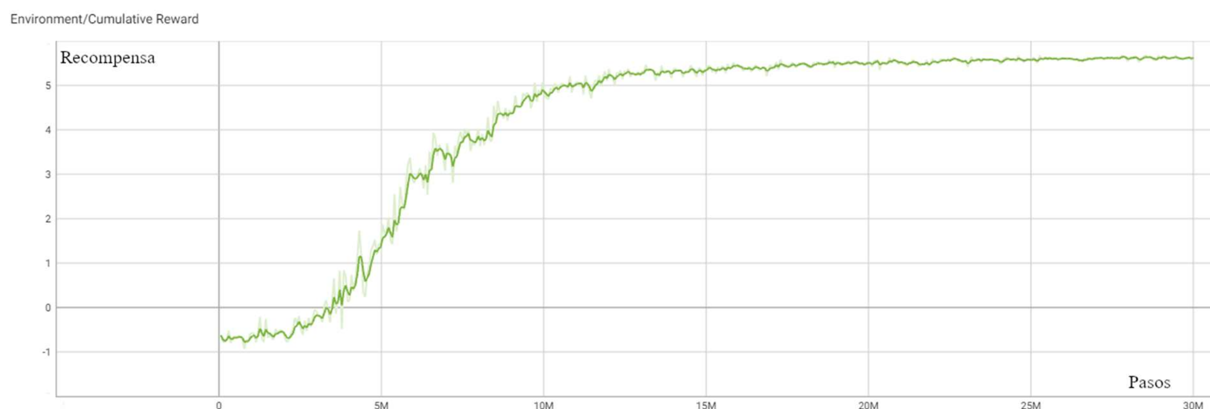

Figura 4.9: Recompensa acumulada en un entrenamiento del entorno 2Bricks.

La recompensa obtenida por situar dos bloques dentro del objetivo es 6 – una recompensa parcial de 2 al colocar cada bloque y una recompensa extra de 2 tras completar la tarea. Otorgar recompensas por subtareas es fundamental, especialmente cuando más se complica la tarea principal. En la gráfica de la *Figura 4.9* se observa que la recompensa acumulada se estabiliza cerca del valor deseado, pero con una diferencia mayor que antes. Observando varios episodios con el modelo entrenado pudo comprobarse que se debe a dos causas:

- De media, los episodios son mucho más largos, aumentando la recompensa negativa que se acumula con cada decisión.
- Hay un pequeño porcentaje de episodios que no se completan satisfactoriamente. Esto sucede cuando se alcanza el máximo de pasos, una constante definida en Unity con un valor muy superior a lo necesario. Su objetivo es evitar que los episodios que se queden atascados continúen indefinidamente.

Los diferentes cambios planteados a partir de este entorno fueron, por separado, añadir un tercer bloque y reducir la anchura del objetivo al ancho de dos bloques (más un pequeño margen). Ambos casos resultaron en entrenamientos fallidos con varias configuraciones de hiperparámetros. Evaluando las posibles causas del problema, se llegó a la conclusión de que la complejidad de la tarea era demasiado alta. Siendo una tarea relativamente sencilla en comparación con el objetivo inicial de apilar bloques para formar un muro, la solución escogida fue simplificar el entorno. Simplificar significativamente el entorno reduce el número de estados drásticamente, facilitando la solución del PDM – que deberá tener objetivos mucho más complejos que el actual.

4.3. ENTORNO DISCRETO 3D

El planteamiento del entorno cambia drásticamente. Pese a la aparente simplicidad de los primeros entornos, la libertad total de movimiento – tanto posición como rotación – del agente y los bloques crea una cantidad de estados diferentes demasiado alta. A partir de este momento, el entorno se contempla como un espacio discreto (ver *Figura 4.10*), en el que el agente solo puede situarse en ciertas posiciones, como si se tratase de un tablero de ajedrez, siendo las únicas posiciones válidas el centro de las casillas. El movimiento de los bloques no se limita, ya que su comportamiento siguiendo físicas realistas se considera importante. Los choques entre bloques pueden desplazarlos fuera del centro de las casillas, pero por su tamaño siempre son detectables por los sensores y el agente puede agarrarlos.

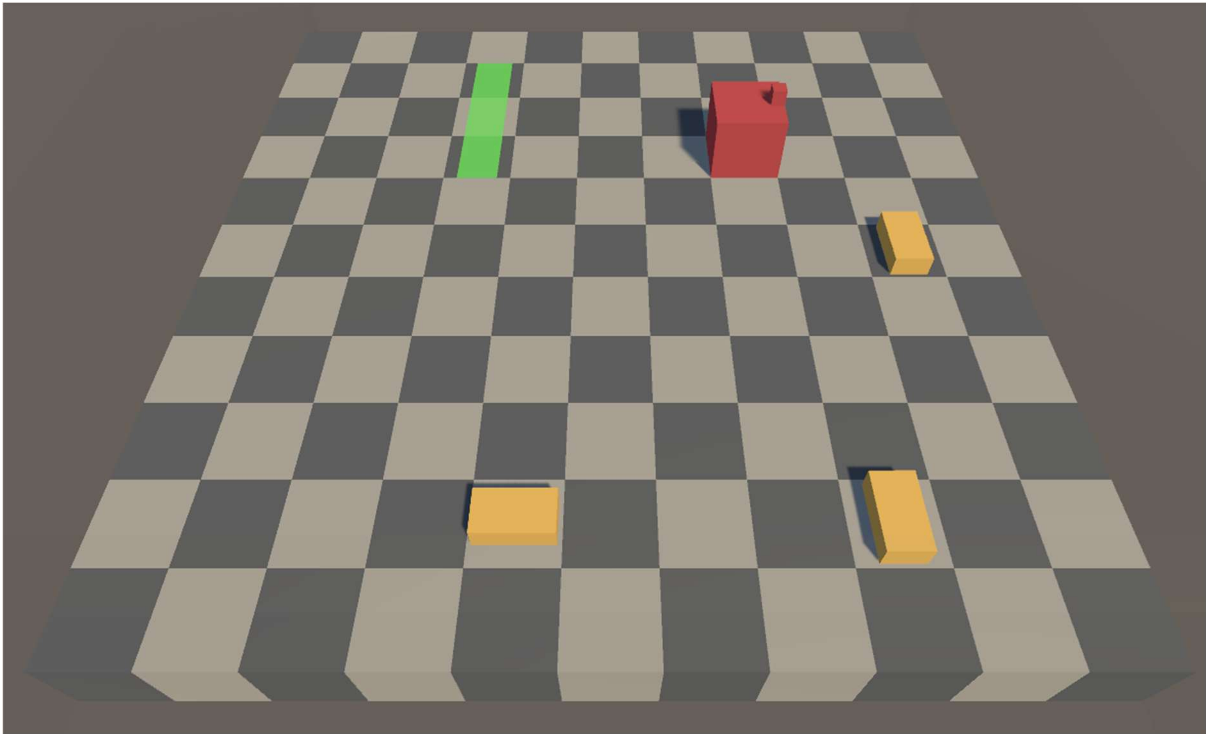

Figura 4.10: Primer entorno discreto.

4.3.1. Discrete

El nuevo planteamiento se implementa realizando los siguientes cambios respecto al entorno continuo:

- Movimiento del agente. En el planteamiento anterior, el movimiento se basaba en la aplicación de fuerzas sobre el *rigidbody* del agente. En este nuevo planteamiento, el agente se mueve directamente sin el uso de físicas realistas. Cada movimiento cubre una distancia de 1 unidad, igual al tamaño del agente. La implementación de la rotación también cambia, pero su funcionamiento es similar: el agente siempre está orientado en el sentido de su último movimiento.
- La toma de decisiones del agente debe coordinarse con el movimiento del agente, que solo debe ser capaz de iniciar una acción cuando está en el centro de una casilla y no mientras se mueve entre ellas. La frecuencia de solicitud de decisiones del agente se limita sustituyendo el componente *DecisionRequester* con una llamada manual a la función *RequestDecision* dentro del método *FixedUpdate* – que, por defecto, se ejecuta cada 0,02 segundos. Situando la llamada dentro de un bloque condicional que comprueba la posición del agente se logra la funcionalidad deseada.
- Se añaden más sensores para cubrir un área más amplia (ver *Figura 4.11*). Se busca un equilibrio entre pocos sensores con la posibilidad de perder información relevante y demasiados sensores que aporten únicamente información redundante. El problema

de utilizar muchos sensores es que requiere emplear redes neuronales más grandes en el entrenamiento, lo que ralentiza el proceso.

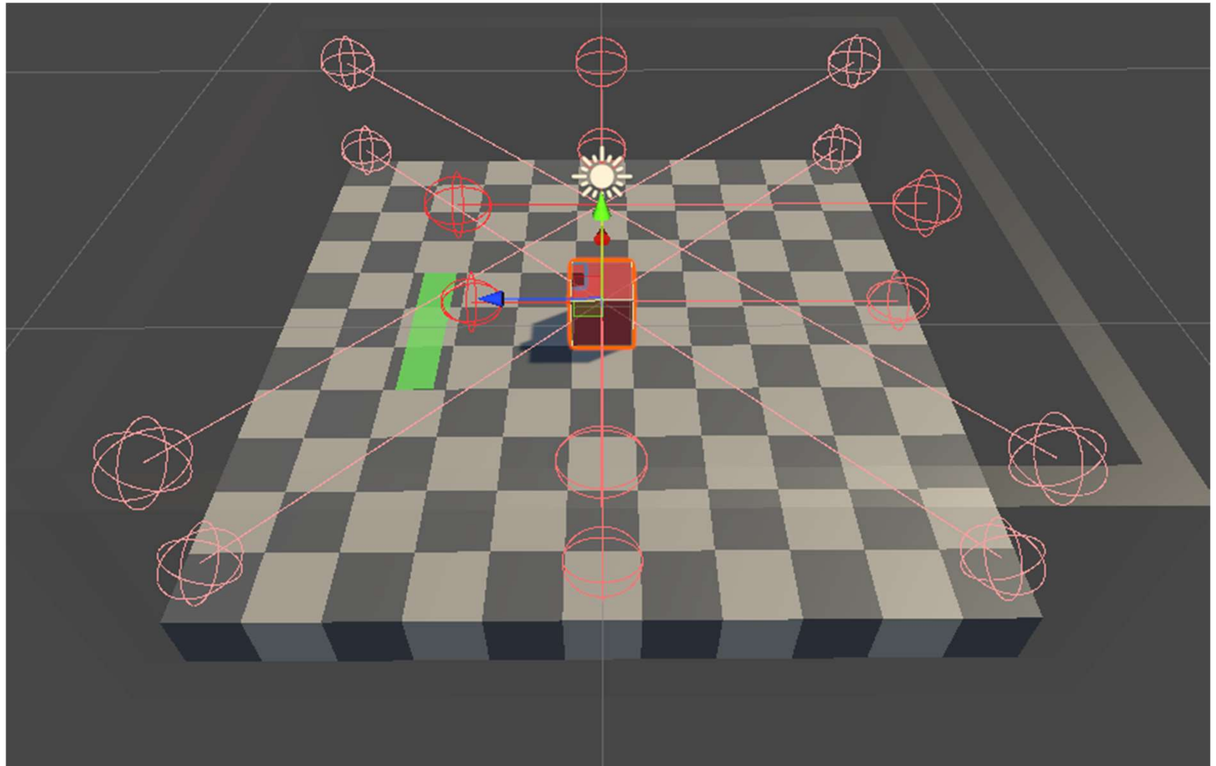

Figura 4.11: Sensores actualizados.

- El área se adapta para que sus dimensiones sean divisibles por la distancia de desplazamiento del agente (1 unidad). La apariencia se cambia, dividiendo la superficie en casillas cuadradas de 1 unidad de lado. Este cambio es meramente visual, para facilitar la depuración durante el desarrollo.
- El tamaño de los bloques se adapta para que ocupen una casilla de anchura. Se deja un margen de 0,05 unidades a cada lado para evitar choques y desplazamientos no deseados que se producían al yuxtaponer bloques.
- Tras los primeros entrenamientos exitosos, se integran los movimientos previamente descartados: rotar, levantar y bajar el bloque agarrado. Rotar nunca es estrictamente necesario, pero sirve para ahorrar varios movimientos del agente; y los otros dos no serán necesarios hasta que se quieran apilar varios bloques. No obstante, se quisieron realizar pruebas con ellos de cara a los próximos objetivos.

La meta que se persigue en esta escena es colocar todos los bloques disponibles dentro de la zona objetivo.

Entrenamiento y resultados

En los primeros entrenamientos había tres bloques que colocar. La recompensa total de la tarea es 9 – 3 por colocar cada bloque –, a la que se resta la penalización de cada decisión.

Las gráficas de recompensa media de diferentes entrenamientos (ver *Figura 4.12*) se estabilizan en valores cercanos a 8. Para comprobar si este valor es correcto, es decir, si la diferencia hasta 9 se debe a la recompensa negativa acumulada a lo largo del episodio o si hay una cantidad significativa de episodios sin éxito, se consulta el gráfico con la distribución de recompensas (ver *Figura 4.13*).

Figura 4.12: Recompensa acumulada en entorno Discrete.

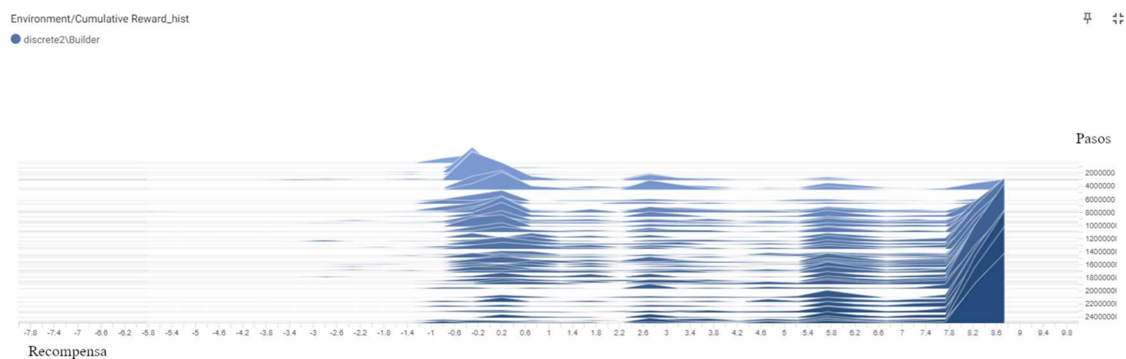

Figura 4.13: Distribución de la recompensa acumulada del mejor entrenamiento del entorno Discrete.

La gráfica de la *Figura 4.13* representa la distribución de recompensas obtenidas por los agentes a lo largo del entrenamiento. En el eje horizontal se muestra la recompensa acumulada por el agente al final del episodio. En el eje vertical se muestra el progreso del entrenamiento. Los primeros episodios están en la parte superior y el final del entrenamiento corresponde a la línea inferior. En cada línea se representa la distribución de la recompensa obtenida por los agentes. En este caso, es notable que al final del entrenamiento la mayoría de los agentes logran una recompensa entre 7,8 y 8,7, que corresponde a haber colocado tres bloques. También se distinguen, a lo largo del entrenamiento, otras tres aglomeraciones de

agentes, correspondientes a la mayoría de los episodios que terminan con 0, 1 y 2 bloques colocados.

4.3.2. Discrete15

Este entorno es el mismo que el anterior, pero con 15 bloques. Por tanto, la recompensa máxima es 45. El objetivo buscado es que el agente consiga apilar bloques de forma fiable.

La recompensa obtenida no se acerca en absoluto al objetivo (ver *Figura 4.14* y *Figura* 4.15). viendo el comportamiento del modelo entrenado se observa que, aunque habitualmente se coloquen varios bloques en la zona, en algún momento suele derribarse parte de lo construido, perdiendo la recompensa acumulada. A continuación, el agente vuelve a colocar algún bloque y vuelve a derribar el muro, ya sea chocándose directamente contra él o chocando el bloque agarrado contra bloques ya colocados. La recompensa media al final del entrenamiento tiende a 15, lo que indica que al final de los episodios hay, de media, 5 bloques correctamente colocados.

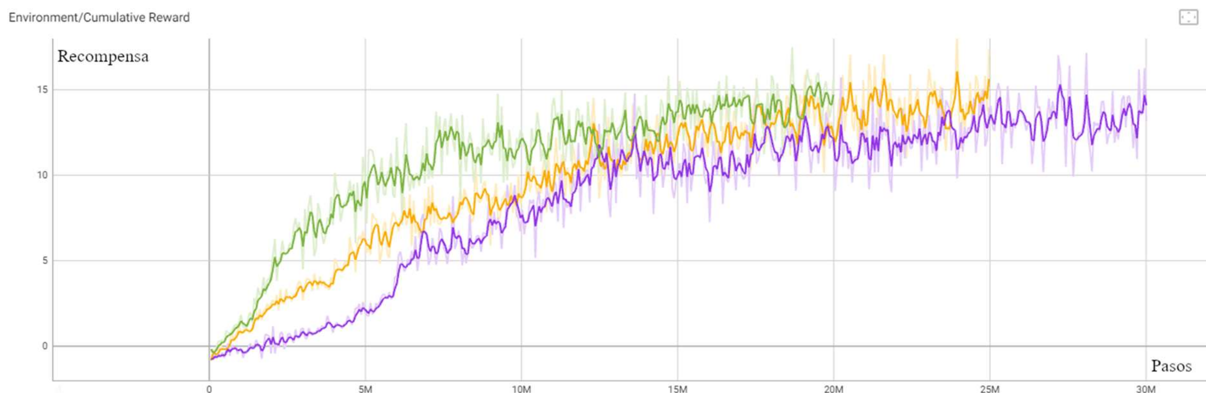

Figura 4.14: Recompensa acumulada en los entrenamientos de Discrete15.

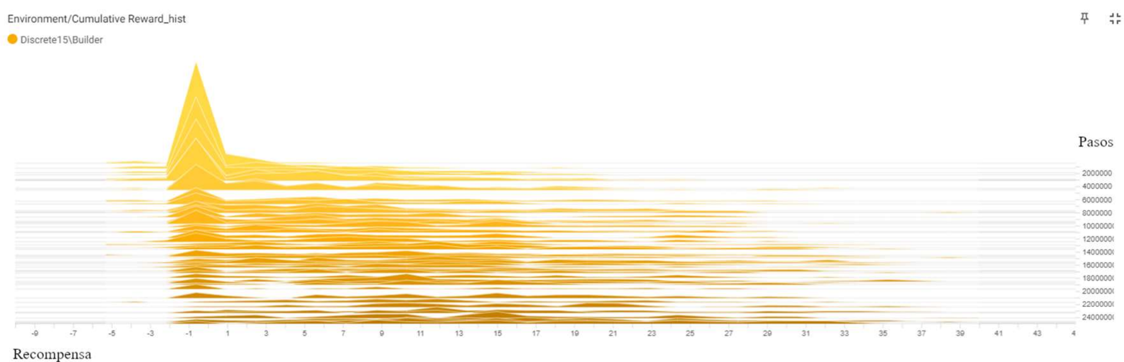

Figura 4.15: Mejor entrenamiento de Discrete15.

Ante el fracaso de este entorno, se cambia el objetivo. Al añadir un número tan elevado de bloques se buscaba que el agente aprendiera a apilar bloques. La tarea ha sido demasiado

compleja para el entrenamiento o no se han planteado correctamente las recompensas parciales para lograr el objetivo. A continuación, se plantea el objetivo de apilar bloques de forma diferente.

4.3.3. DiscreteSmall

Partiendo del entorno *Discrete*, se disminuye la anchura de la zona objetivo a 1 unidad de ancho y la cantidad de bloques para a ser 7. Para colocar los bloques en la zona, el agente deberá apilarlos, siendo necesario levantar los últimos bloques para no tirar los anteriores. Es decir, en los estados similares al representado en la *Figura 4.16*, el agente debe aprender que la secuencia de acciones con mayor retorno es levantar el bloque, avanzar y soltarlo dentro de la zona.

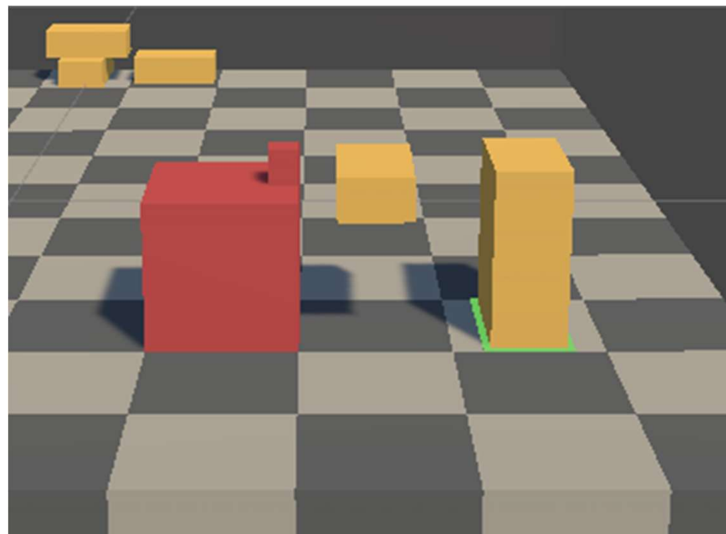

Figura 4.16: Estado clave para la construcción en vertical.

Este intento de hallar una solución para la construcción en vertical tampoco tiene éxito. La recompensa media al final del entrenamiento tiende a 8, muy lejos del objetivo, que es 17,5 (ver *Figura 4.17*). Integrando el modelo entrenado en Unity se observó que el agente no había aprendido a solucionar el estado clave mostrado en la *Figura 4.16*, derribando parte del muro construido con el nuevo bloque que intenta colocar.

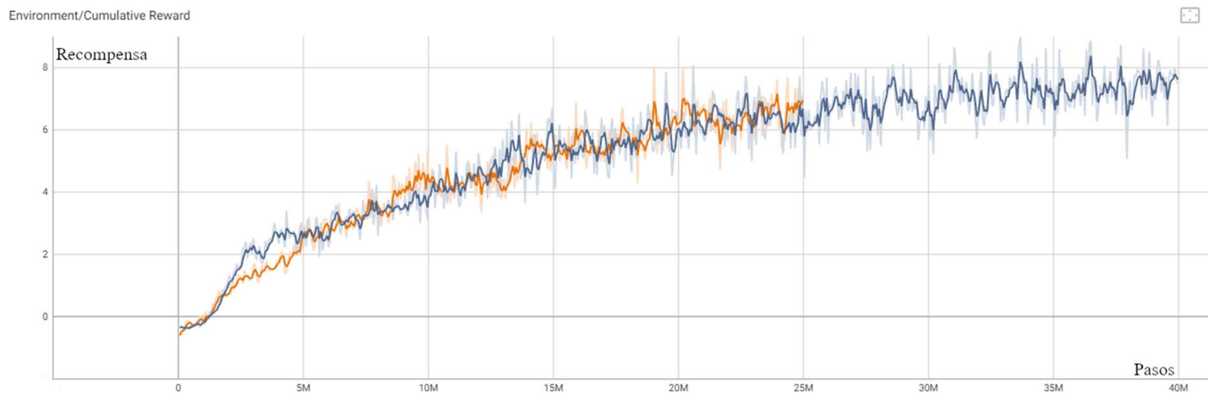

Figura 4.17: Recompensa acumulada en los entrenamientos de DiscreteSmall.

4.3.4. DiscreteV2

El mayor problema del entorno anterior es que el agente tiende a destruir lo que previamente ha construido. El agente es capaz de detectar dónde está la zona objetivo, pero pasa sobre ella independientemente de que haya bloques colocados o no. Para intentar solventar este problema, se diseña un nuevo conjunto de 8 sensores, que únicamente detectarán bloques. Orientados frontalmente, están a la distancia adecuada para que cada uno detecte un bloque diferente cuando se apilan. La nueva configuración de sensores puede verse en la *Figura 4.18*. Nótese que los sensores rojos – los únicos que había antes –, detectan el objetivo sin aportar información sobre si hay bloques en él o no.

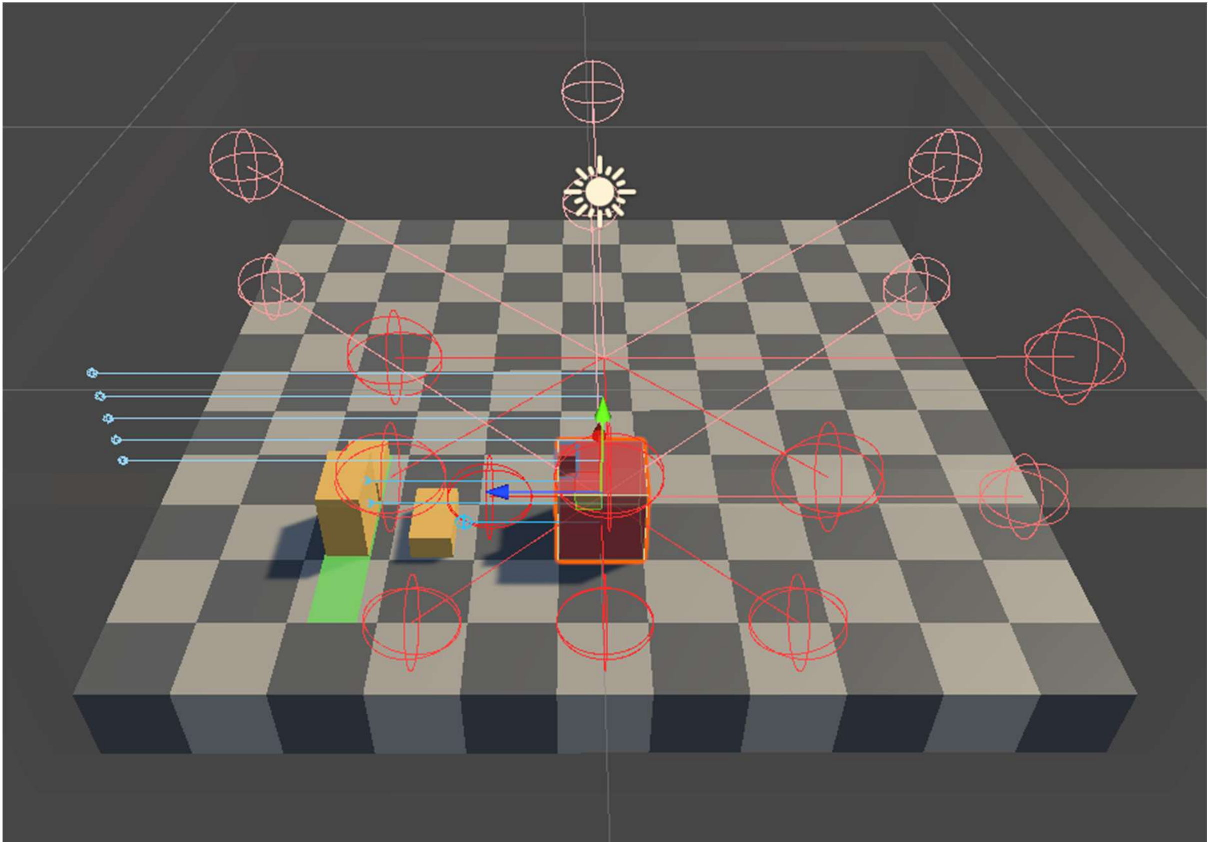

Figura 4.18: Sensores del agente en DiscreteV2. Los nuevos sensores se representan en color azul.

Entrenamiento y resultados

El primer objetivo planteado en el nuevo entorno es colocar 10 bloques dentro de la zona objetivo, de 3 unidades de anchura.

El entrenamiento no llega a aprender completamente, pero se observa un progreso continuo, en la *Figura 4.19* y especialmente en la *Figura 4.20*, correspondiente al mejor entrenamiento. Incrementando bastante la longitud del entrenamiento se podría obtener un buen resultado. En vez de emplear tanto tiempo, se decide entrenar el mismo entorno con menos bloques, considerando que la tarea es la misma pero los episodios serán significativamente más cortos.

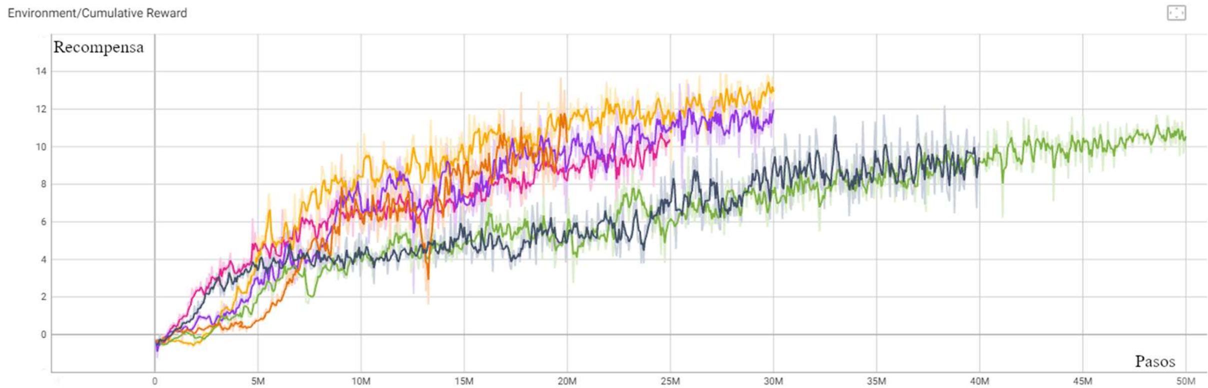

Figura 4.19: Recompensa acumulada en los entrenamientos de DiscreteV2.

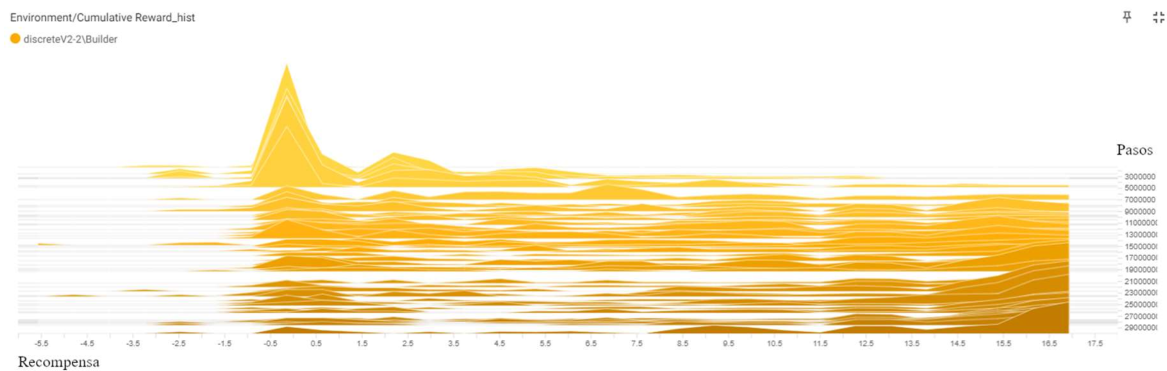

Figura 4.20: Mejor entrenamiento de DiscreteV2.

4.3.5. dV2-5B

Los siguientes entrenamientos se realizan en el mismo entorno, reduciendo el número de bloques a 5. La recompensa máxima en este caso es 12,5 (ver *Figura 4.21*).

Figura 4.21: Recompensa acumulada en los entrenamientos de dV2-5B.

La forma de las curvas de la gráfica corresponde a un aprendizaje satisfactorio, pero la recompensa acumulada alcanzada es levemente menor de lo esperado de un aprendizaje completamente exitoso.

Por primera vez se emplea una métrica hecha a medida para supervisar el entrenamiento: pasos por muro completo (ver *Figura 4.22*). Esta métrica representa el número de pasos que transcurren desde el comienzo de un episodio hasta que se completa con éxito. Debe notarse que los episodios fallidos no cuentan, por lo que no representa realmente cómo de bueno es el agente hasta que la recompensa media no se estabilice cerca del valor óptimo.

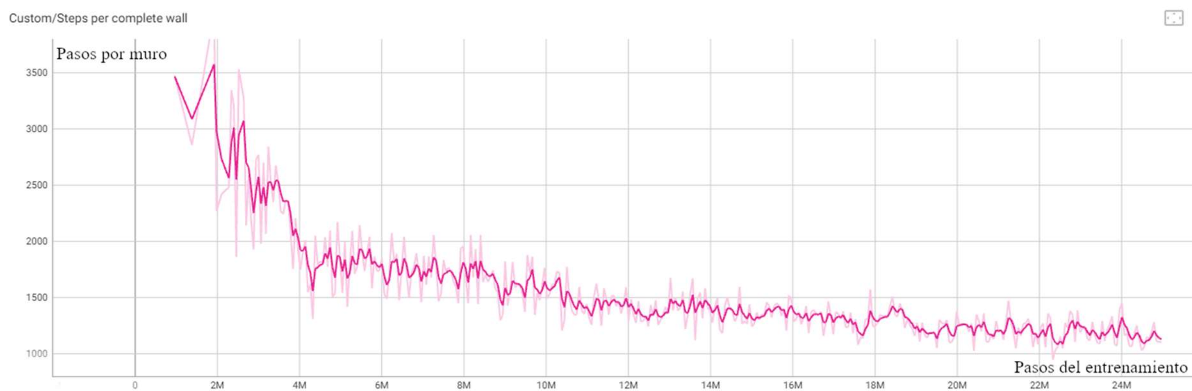

Figura 4.22: Pasos por muro completo en dV2-5B.

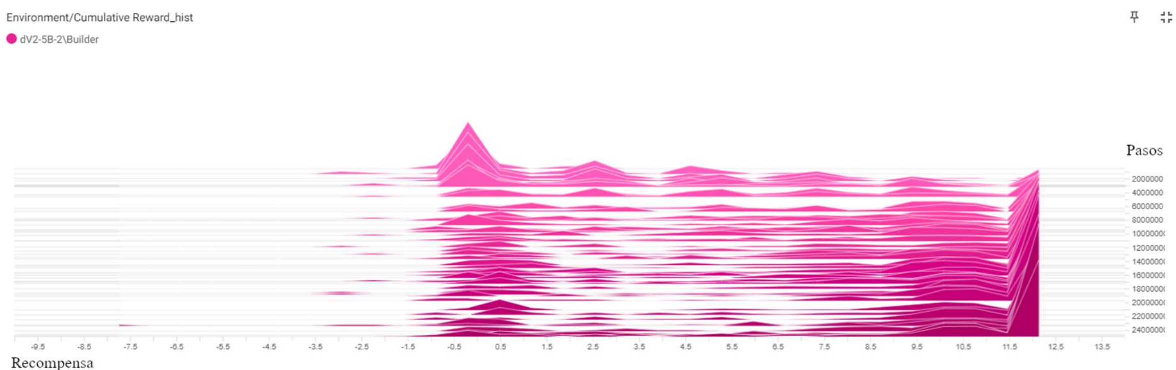

Figura 4.23: Distribución de recompensa acumulada en dV2-5B.

La distribución de recompensas obtenida al final del entrenamiento, representada en la parte inferior del gráfico de la *Figura 4.23*, muestra que la gran mayoría de los episodios obtienen una recompensa correspondiente al éxito total. Sin embargo, hay una pequeña porción de episodios que no terminan adecuadamente, con una recompensa media cercana a 10,5. Ejecutando varios episodios con el modelo entrenado se observa fácilmente lo que sucede en estos episodios: el agente coloca de forma correcta cuatro bloques, pero nunca detecta el último. En vez de explorar toda el área de una forma organizada como haría una persona, el agente tiende a repetir el mismo ciclo de movimientos, dejando zonas del entorno fuera del

alcance de sus sensores. Esto suele pasar cuando el último bloque está cerca de una esquina – zonas que los sensores cubren peor.

El movimiento del agente no es eficiente: es capaz de desplazarse frente a un bloque y agarrarlo con relativa fluidez, pero sus movimientos suelen tornarse erráticos cuando debe ir a colocar el bloque, especialmente cuando hay objetos entre el agente y el objetivo, obstaculizando los sensores.

Durante la revisión del modelo entrenado, también se observa que el agente sigue derribando el muro ocasionalmente, aunque con menor frecuencia que antes de añadir los sensores frontales. La probabilidad de derribar el muro crece considerablemente cuanto más largo es el episodio, lo que complica el aprendizaje de las tareas asociadas a la colocación de los últimos bloques. Esta puede ser la causa del aprendizaje relativamente correcto obtenido en general, pero muy deficiente en lo relativo a una tarea que sólo se da en episodios largos.

4.3.6. DiscreteV3

Ante los pocos avances logrados desde el primer entorno discreto, se crea una nueva escena con una serie de cambios sobre el entorno *DiscreteV2* que buscan dar solución a los problemas presentes:

- En primer lugar, se reduce el tamaño del área a 7x7 casillas. Cuanto más pequeño y más sencillo es el entorno, más fácil es que el agente aprenda una buena política. Cuando se obtengan resultados favorables en este entorno, se tratarán de replicar en entornos más grandes.
- Se impide que el agente derribe el muro. Para lograrlo, se llevan a cambio dos modificaciones:
    - En el código que se encarga del movimiento del agente se impide que el agente entre en la zona objetivo. Esta restricción es una manera de influir de una forma bastante directa en el PDM. Lo que se está haciendo es modificar el conjunto de acciones posibles en ciertos estados.
    - Los bloques que se colocan dentro de la zona quedan fijados. De esta forma se evita que el bloque agarrado por el agente pueda derribar bloques del muro al chocarse con ellos. Esta funcionalidad y las comprobaciones necesarias para su funcionamiento correcto se implementan añadiendo dos métodos al componente *DisPickable.cs* de los bloques (sucesor de *Pickable.cs*). Ambos métodos son llamados cuando un bloque no colocado está dentro de la zona objetivo.
        - El método *isGrounded* comprueba, mediante un *raycast*, si está apoyado sobre el suelo o si se trata de un bloque etiquetado como colocado.
        - El método *inPlace* elimina el *rigidbody* del bloque, haciendo imposible interactuar con él.

- Se divide el *trigger* de la zona objetivo por niveles, de forma que cada bloque apilado esté dentro de un nivel (ver *Figura 4.24*). De esta forma, se abren nuevas posibilidades en la forma de dar recompensas, permitiendo definir nuevas tareas y subtareas (por ejemplo, completar un nivel). También se pueden asignar recompensas diferentes a la colocación de un bloque en distintos niveles para intentar priorizar la construcción en orden.

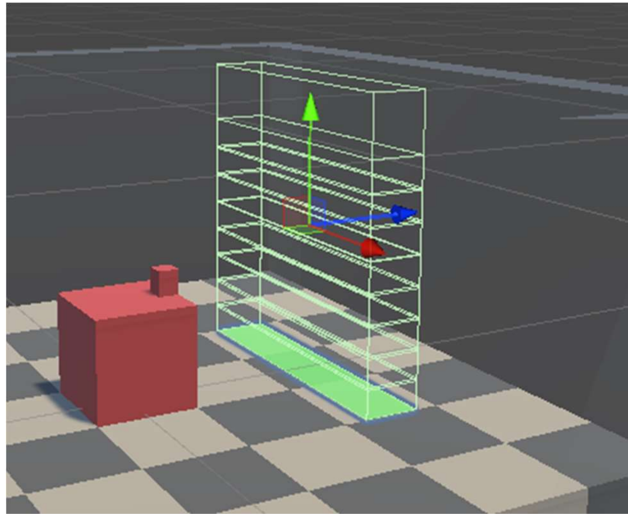

Figura 4.24: Zona objetivo con trigger con niveles.

Entrenamiento y resultados

Los primeros entrenamientos se realizaron con un sistema de recompensas que otorga una recompensa por cada bloque colocado (su valor es 8 en el nivel inferior y decrece en una unidad cada nivel) y una recompensa extra de valor 10 al completar un nivel de 3 bloques de anchura (ver *Figura 4.25*). Cada episodio se inicializa con 6 bloques y se añade una métrica nueva a TensorBoard: número de pasos por nivel completado, con un funcionamiento equivalente a pasos por muro completado (ver *Figura 4.26*).

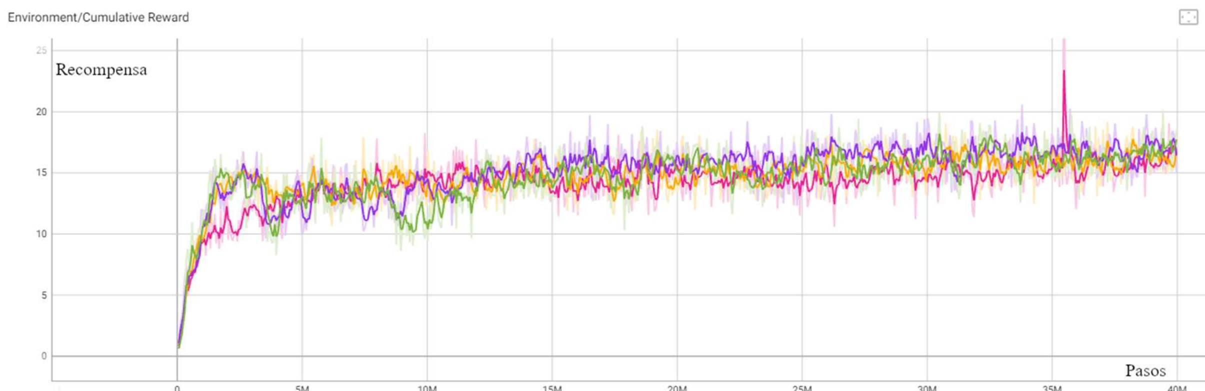

Figura 4.25: Recompensa acumulada en algunos entrenamientos de DiscreteV3.

Los resultados obtenidos en los mejores entrenamientos son peores de lo esperado. La recompensa media corresponde a la colocación de varios bloques, pero no hay ningún indicio de que el agente busque la recompensa de completar un nivel. Observando el comportamiento de algunos de los modelos entrenados pudo comprobarse que las políticas obtenidas son muy malas: el movimiento del agente sigue siendo muy errático.

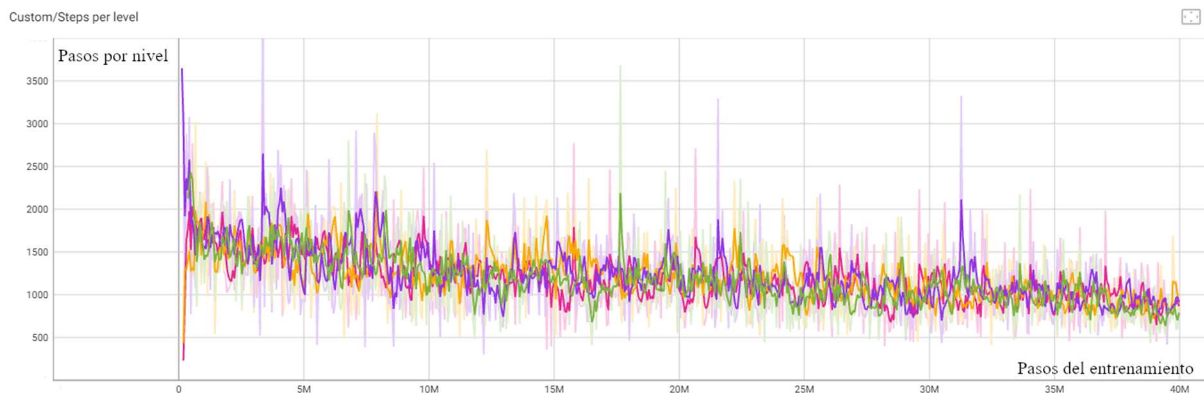

Figura 4.26: Pasos por nivel en entrenamientos de DiscreteV3.

La nueva gráfica que monitoriza el número de pasos empleados para completar un nivel corrobora que no hay ninguna mejora aparente a lo largo del entrenamiento (ver *Figura 4.26*). La gran cantidad de ruido en los datos se debe a que hay pocos episodios representados – únicamente los que consiguen completar un nivel –, una muestra más del bajo porcentaje de éxito.

Al no encontrar una causa aparente en el entorno a la falta de eficiencia de los entrenamientos, y por la incapacidad de mejorar los resultados con otras configuraciones de hiperparámetros, se comienza a desarrollar una nueva funcionalidad del agente para acercarse más a la construcción realista de un muro. Mientras tanto, se continúa buscando el origen del problema. En posteriores *sprints* de desarrollo se descubren varios fallos: los *triggers* por niveles no tienen el suficiente margen, lo que lleva a no detectar a veces los bloques colocados. Además, los sensores no detectan correctamente las paredes que limitan el entorno cuando el agente está en contacto con ellas – las tres parejas de sensores a la izquierda en la *Figura 4.27*, que atraviesan una pared que deberían detectar.

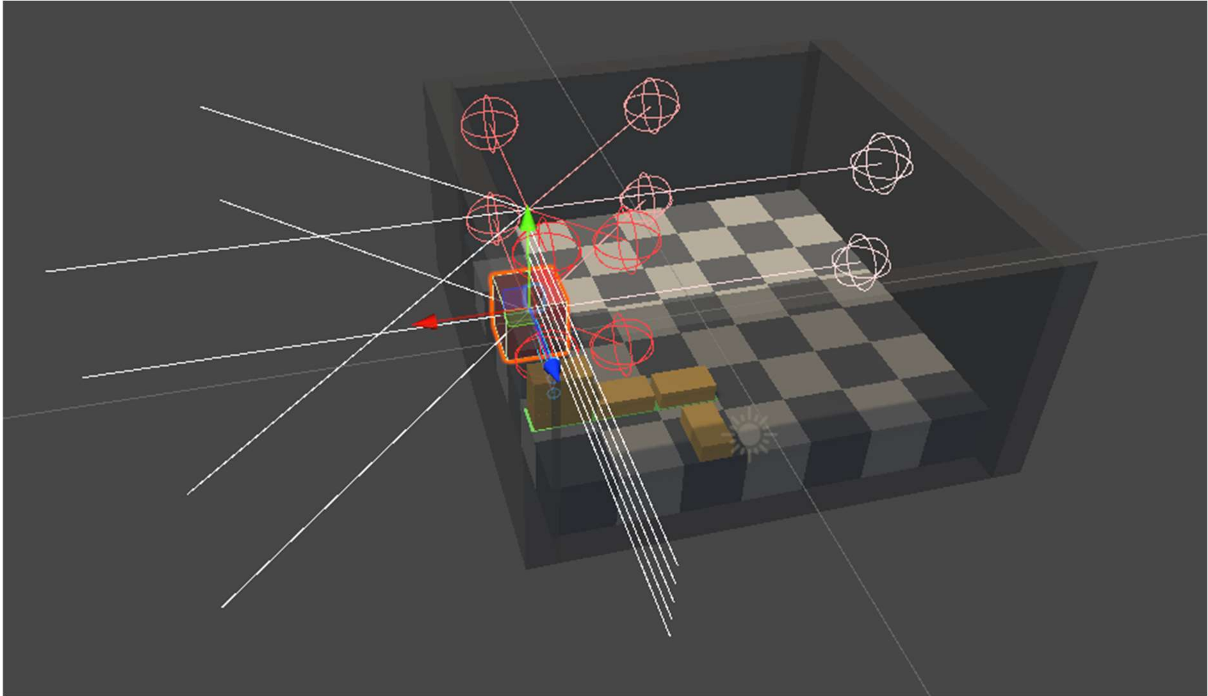

Figura 4.27: Sensores funcionando incorrectamente.

4.3.7. DiscreteV4

En los entornos discretos anteriores, el agente podía apilar cada bloque directamente encima del anterior. Con esta actualización del agente se quiere permitir una técnica de construcción más robusta y realista.

Se denomina aparejos a las diferentes formas de colocar las piezas de construcción en una pared. Se ha escogido el aparejo de soga (ver *Figura 4.28*) [37] como la técnica que el agente será capaz de utilizar, por su sencillez y parecido con el entorno desarrollado hasta el momento.

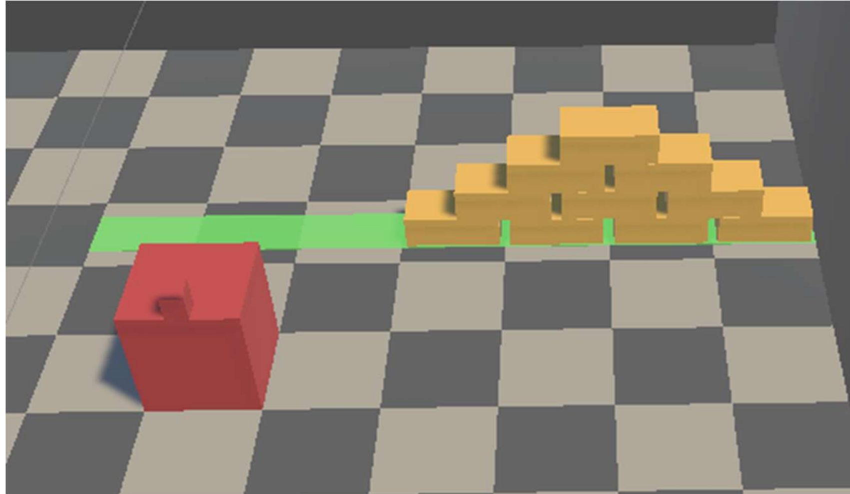

Figura 4.28: Bloques colocados según el aparejo de soga.

Con este fin, se implementa una nueva acción al agente. La acción consiste en desplazar el bloque agarrado una distancia de media casilla lateralmente respecto al agente (ver *Figura 4.29*). Si se repite la acción, se devuelve el bloque a su posición original.

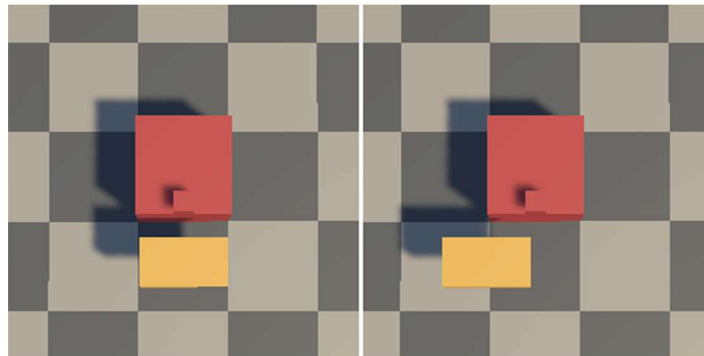

Figura 4.29: Movimiento lateral de un bloque.

La colocación de bloques en aparejo de soga no funciona correctamente tras añadir la nueva acción. Para fijar un bloque en el muro (y obtener una recompensa), deben suceder los siguientes eventos:

1. El bloque está dentro de un nivel del *trigger* del objetivo, disparando el método *OnTriggerStay* del componente *GDetectV4* (sucesor de *GoalDetect* que detecta colisiones). Este método llama al método del agente *BrickInPlace*, pasando como parámetro el identificador del nivel.
2. Dentro de *BrickInPlace*, se comprueba si el bloque está apoyado utilizando el método *isGrounded* del componente *DisPickable* del bloque.

3. Si la comprobación es correcta, se llama al método que fija al bloque – *inPlace*, también del componente *DisPickable*.

El problema se encuentra en el segundo paso. El método *isGrounded* lanza un *raycast* hacia abajo desde el centro del bloque. Al colocar los bloques de forma alterna, el *raycast* no detecta el suelo o un bloque inmediatamente debajo. La solución adoptada es sustituir el *raycast* centrado por dos *raycasts* situados a ambos lados del centro. Es necesario que ambos detecten un bloque o el suelo para considerar al bloque como colocado. En la *Figura 4.30* se comparan de forma esquemática el *raycast* antiguo, en rojo, y los *raycasts* nuevos, en verde.

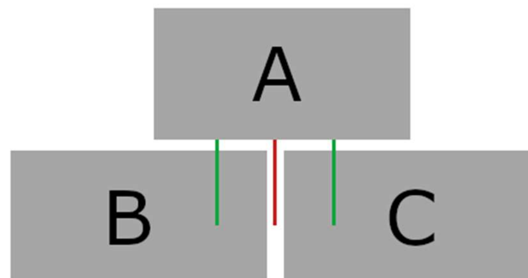

Figura 4.30: Raycasts lanzados por el bloque A a los bloques B y C (raycast antiguo en rojo y raycasts nuevos en verde).

El *raycast* antiguo (color rojo) ahora se emplea en un nuevo método que detecta si un bloque se ha colocado según el aparejo de soga o no, para proporcionar una recompensa adicional.

Tras un primer bloque de entrenamientos, también se modifican la cantidad sensores del agente. El número de sensores empleados en detectar el entorno – bloques, objetivo y paredes – aumenta de 8 a 16. Cada uno de los sensores frontales – que detectan bloques dentro del objetivo – pasa a ser un conjunto de 5 sensores distribuidos en un abanico de 90°.

Entrenamiento y resultados

El gráfico de la *Figura 4.31* representa la distribución de recompensas en uno de los mejores entrenamientos. Puede observarse que, aunque se mejora claramente respecto al inicio del entrenamiento, no se consigue aprender a obtener recompensas altas de forma consistente.

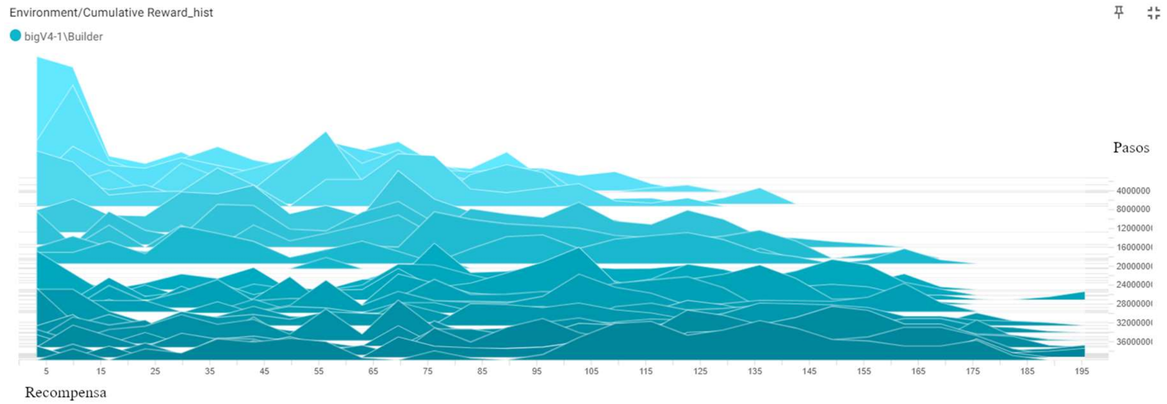

Figura 4.31: Distribución de recompensas en un entrenamiento de DiscreteV4.

Durante la revisión de los resultados se observa:

- No se utiliza la nueva acción. La política aprendida (que sigue teniendo un comportamiento parcialmente errático) no se prioriza. Es un resultado esperado, ya que es una tarea complicada que depende de otra más básica – construir un nivel del muro o parte de él, que aún no se ha conseguido aprender satisfactoriamente.
- El agente puede quedarse bloqueado en posiciones intermedias entre casillas hasta que el episodio alcanza su duración máxima y termina automáticamente (ver *Figura 4.32*). Es un problema que puede suceder al mover el agente hacia una de las paredes que limitan el entorno cuando tiene un bloque agarrado.

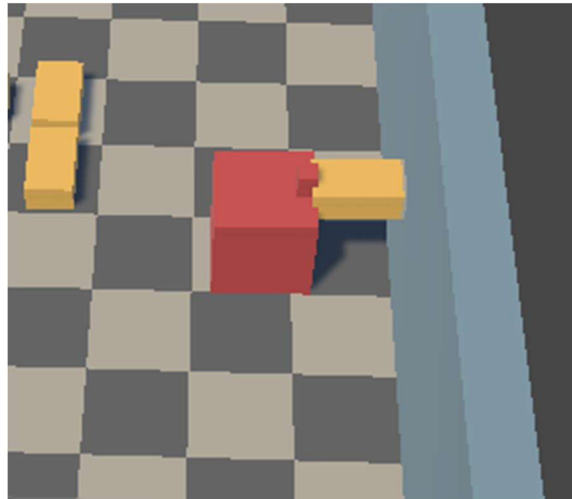

Figura 4.32: Agente bloqueado.

4.3.8. DiscreteV5

La nueva versión del entorno busca proporcionar mejoras en dos aspectos: dar solución a los problemas encontrados en el diseño del entorno y plantear un nuevo enfoque más sencillo para el empleo del aparejo de soga en la construcción.

Cambios en el entorno:

- El principal cambio del entorno es la adición de un margen entre el área por la que se mueve el agente y las paredes (ver *Figura 4.33*). El margen, de 1 unidad de anchura, no forma parte del área por la que el agente se mueve y donde se generan la zona objetivo y los bloques. Los bloques, sin embargo, sí pueden ser empujados o depositados sobre él. Este cambio soluciona completamente los problemas de los sensores que en ocasiones atravesaban las paredes y del agente que se quedaba bloqueado entre dos casillas.

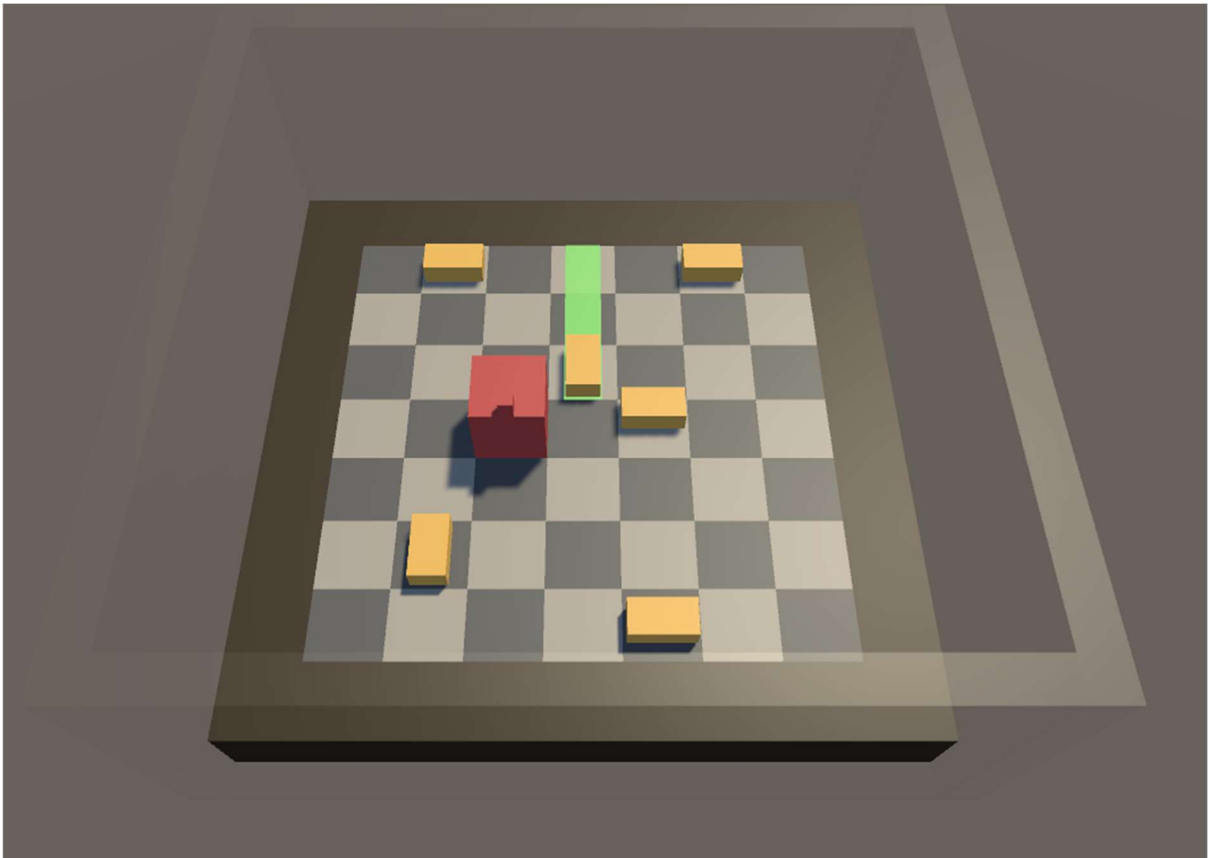

Figura 4.33: Margen entre el tablero y las paredes.

- También se descubre un fallo en la funcionalidad de agarrar un bloque, que, tras revisar los entornos anteriores, se comprueba que ha existido desde el principio. En muy raras ocasiones, cuando el agente se encuentra en contacto con un bloque e intenta agarrarlo, el bloque no es detectado por el *raycast* del método *grabBrick*. La

solución que se encuentra es mover el origen del *raycast* levemente hacia el interior del agente (0,05 unidades), haciéndolo capaz de detectar objetos que se encuentren justo en la superficie del agente. Para mantener el rango del agarrado de bloques, se aumenta la longitud del *raycast* 0,05 unidades.

- Por último, se modifica el comportamiento de un bloque al ser agarrado, para no bloquear parte de los sensores dedicados a detectar el muro. Para no perder información, se añade un nuevo sensor – un componente *Ray Perception Sensor 3D* – que únicamente detecta si el agente tiene un bloque agarrado.

Implementación del aparejo de soga:

- Se elimina la acción de desplazamiento de un bloque lateralmente introducida en el entorno *DiscreteV4*. Para permitir la colocación de bloques cada media casilla, se reduce la distancia que recorre el agente en cada movimiento a la mitad (ver *Figura 4.34*).

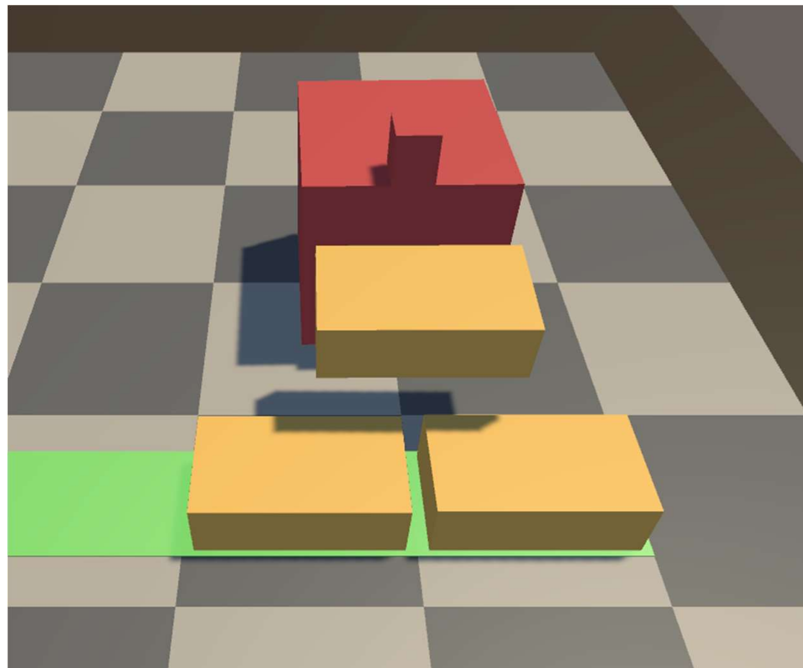

Figura 4.34: Movimiento en distancias de media unidad.

El desarrollo del entorno tridimensional hasta llegar a esta versión, que es la última, ha conllevado una parte del tiempo y esfuerzo mucho mayor de lo esperado al comienzo del proyecto, cuando apenas tenía conocimiento sobre Unity. Hallar e implementar soluciones a los problemas que se van encontrando es un proceso largo en el que, no raramente, la primera solución programada crea nuevos problemas, siendo necesario solucionarlos o desechar el trabajo realizado.

Un buen ejemplo de este proceso es cómo se intentó solucionar el problema del agente atascado. En primer lugar, la mejor solución parecía deshabilitar el *rigidbody* de los bloques

al agarrarlos y volver a habilitarlo cuando se soltaran. Esta solución funciona perfectamente, pero daba la posibilidad de obtener reiteradamente una recompensa al soltar y volver a coger un bloque dentro del objetivo (cada vez que se reactiva el *rigidbody*, se obtiene la recompensa por entrar al muro, pero no se retira la recompensa porque el *rigidbody* nunca sale de él). Tras intentar que funcionara correctamente de varias maneras, hubo que desechar la idea y se desarrolló la solución de añadir un margen entre el área y las paredes, descrita anteriormente.

Entrenamiento y resultados

Los primeros entrenamientos suponen una mejora respecto a los entornos anteriores. El movimiento del agente es mucho menos errático, siendo relativamente eficiente en la tarea de recoger bloques y depositarlos en el objetivo. No obstante, no se consigue que el agente priorice las recompensas de las tareas más complejas – niveles, colocación en aparejo de soga cuando sea posible.

Tras los primeros entrenamientos, se prueba a utilizar la opción *stacked vectors*. Esta opción permite al agente utilizar observaciones recogidas a lo largo de varios pasos en vez de contar únicamente con la observación recogida por los sensores en el paso más reciente. La función de esta opción es capturar información que depende del tiempo, como la velocidad del movimiento un objeto. La contrapartida que tiene es que su uso multiplica el número de datos de entrada de la red neuronal, pudiendo requerir una de mayor tamaño que ralentice el entrenamiento.

Al utilizar la opción *stacked vectors*, hay una mejora notable empleando las observaciones de los dos últimos pasos. A partir de tres *stacked vectors* no hay ninguna mejora. La mejora se refleja únicamente en un movimiento del agente más eficiente, que sigue sin priorizar la colocación ordenada de bloques en niveles completos o en aparejo de soga.

En las gráficas de las *Figura 4.35* y *Figura* 4.36 se aprecia que el entrenamiento con dos *stacked vectors*, en azul, completa muchos más muros – es decir, coloca todos los bloques en un episodio – que el entorno que no utiliza esta opción. La recompensa media también mejora significativamente.

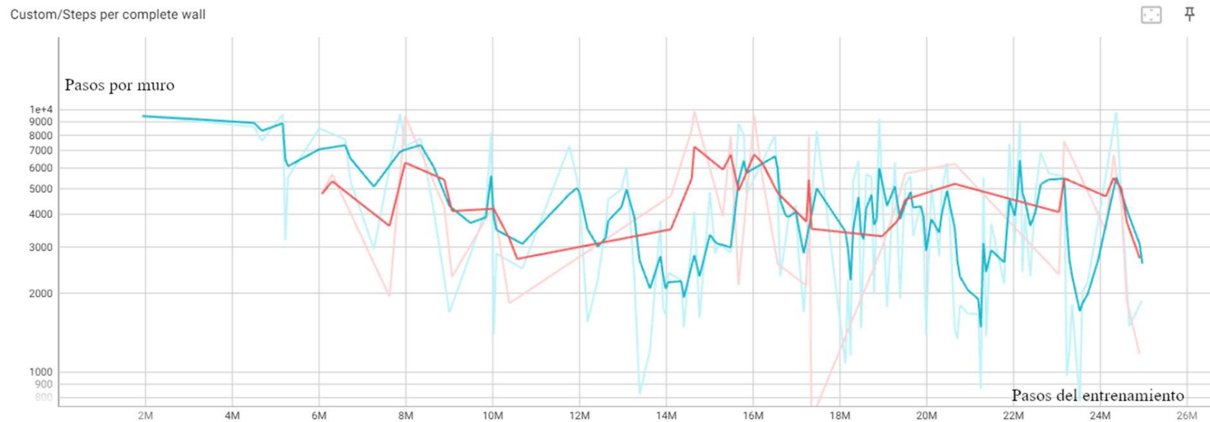

Figura 4.35: Pasos por muro completado. Dos stacked vectors (azul) y sensores originales (rojo).

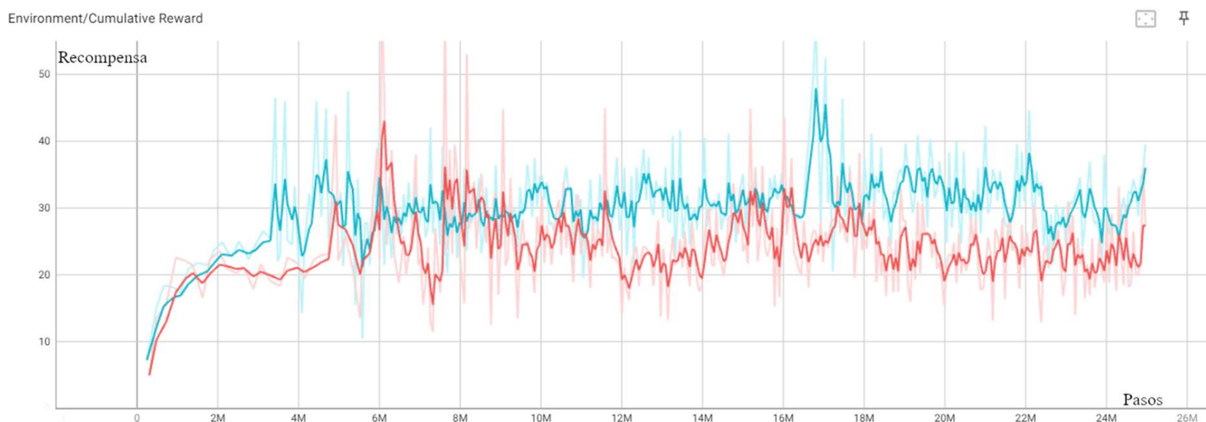

Figura 4.36: Recompensa acumulada. Dos stacked vectors (azul) y sensores originales (rojo).

Ante la falta de más progreso, que ya no se debe parcialmente a fallos en el diseño del entorno, se busca simplificar el PDM drásticamente, cambiando el planteamiento del entorno, como se verá en el próximo apartado. En el nuevo entorno se aprende que el tiempo de entrenamiento necesario para aprender una tarea tan compleja es mucho mayor del empleado hasta ahora, incluso en entornos muy simples.

A partir de lo aprendido en el entorno simplificado, al final del desarrollo del proyecto se vuelve a realizar un último entrenamiento en este entorno. Se configura un entrenamiento con 5 bloques a colocar y 300 millones de pasos, que ha tardado en completarse 4 días, 18 horas, 42 minutos (ver *Figura 4.37*). Este tiempo puede ser adecuado para producir un modelo entrenado como producto final, pero es prohibitivo para los muchos entrenamientos necesarios durante el desarrollo teniendo en cuenta las limitaciones temporales de un Trabajo Fin de Grado.

Los resultados del entrenamiento son claramente mejores que los obtenidos previamente. Desafortunadamente, este entrenamiento no llega a estabilizarse, dejando patente que este entorno requiere entrenamientos aún más largos.

```
[INFO] Builder. Step: 300000000. Time Elapsed: 412922.636 s. Mean Reward: 33.739. Std of Reward: 14.372. Training.
[INFO] Exported results\v5s-june-300M\Builder\Builder-299999945.onnx
[INFO] Exported results\v5s-june-300M\Builder\Builder-300000009.onnx
[INFO] Copied results\v5s-june-300M\Builder\Builder-300000009.onnx to results\v5s-june-300M\Builder.onnx.
```

Figura 4.37: Registro del final del entrenamiento. Pueden verse los segundos transcurridos en la primera línea.

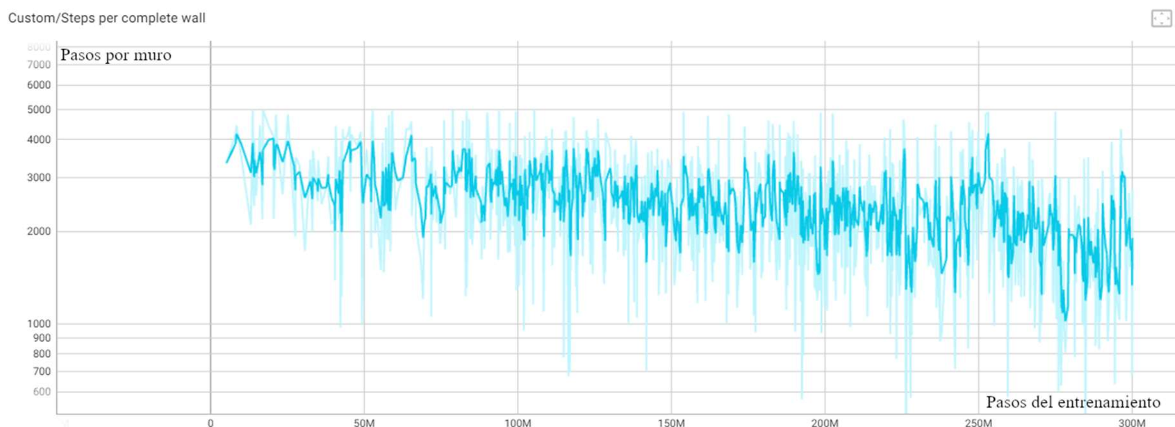

Figura 4.38: Pasos por muro completo en el entrenamiento largo.

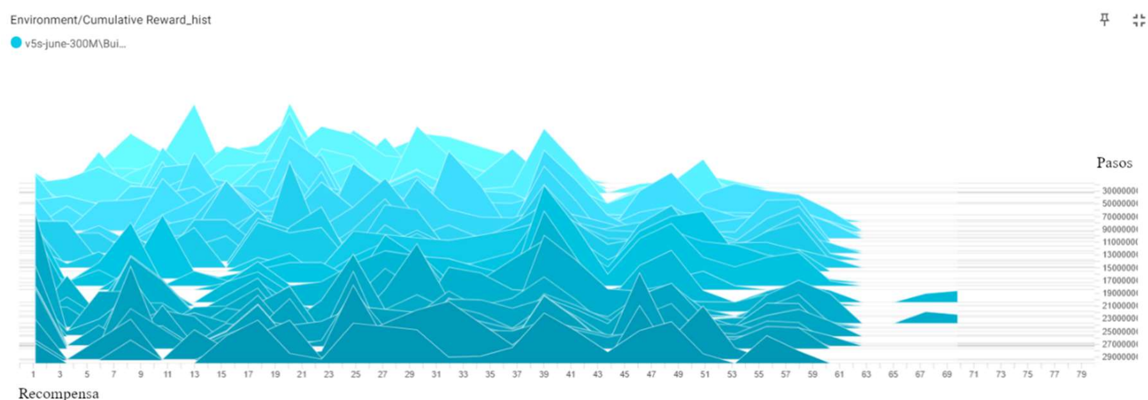

Figura 4.39: Distribución de recompensas en el entrenamiento largo.

En gráficas de las *Figura 4.38* y *Figura* 4.39 se observa un aprendizaje constante, pero que no finaliza. En la primera de ellas (ver *Figura 4.38*), se espera que el número de pasos requeridos para completar un muro se estabilice. En la segunda (ver *Figura 4.39*), se espera que las distribuciones más recientes – en la parte inferior – tengan la mayoría de los episodios agrupados en un único bloque.

4.4. ENTORNO DISCRETO 2D

El planteamiento del último entorno del proyecto supone un cambio drástico respecto a los anteriores. Los dos principios que guían su diseño son:

- El entorno debe ser lo más sencillo posible. Este principio incluye, por ejemplo, la percepción del espacio como un entorno discreto, cambio ya realizado en el movimiento del agente a partir del entorno *Discrete*. Con esta modificación, la posición de los bloques y la información recogida por los sensores serán variables discretas, simplificando el entorno. También se ha considerado eliminar el eje vertical, ya que su único uso es apilar bloques – un concepto que habría que definir de otra manera –, y añade bastante complejidad.
- La percepción del entorno debe ser la máxima posible. Si bien es imposible conocer el entorno y sus dinámicas al 100%, es posible mejorar notablemente la información recogida del entorno respecto a los entornos anteriores para ayudar al algoritmo del aprendizaje a obtener mejores resultados.

Seguir estos dos principios implica crear un entorno con menor realismo. La posición discreta de los bloques impide que éstos se rijan por la física empleada hasta el momento. La percepción del entorno tampoco puede basarse en el punto de vista del agente, sino que se debe utilizar información obtenida por un observador omnisciente.

El diseño ideado a partir de estos dos conceptos es un entorno bidimensional, también organizado en casillas cuadradas. Antes de comenzar a implementarlo, fue necesario el aprendizaje de algunos conceptos de Unity 2D, ya que el nuevo proyecto Unity no se basa en la física de los *rigidbodies* y *colliders* utilizados hasta el momento. Para ello, ha sido necesario recurrir de nuevo a material educativo disponible en la plataforma YouTube. [38]

El entorno se construye sobre un proyecto Unity de código abierto que se explica en uno de los tutoriales disponibles en dicha plataforma [39]. En particular, ese proyecto proporciona los elementos básicos sobre los que se implementará la lógica del agente:

- Creación de cuadrículas formadas por casillas cuadradas. Una cuadrícula representará el área por la que se mueve el agente y donde se encuentran los bloques y la zona objetivo. Las dimensiones de la cuadrícula y el tamaño de cada casilla son fácilmente modificables.
- Capacidad de almacenar un objeto en cada casilla.
- Definición de la clase de los objetos almacenados. La clase empleada es muy sencilla: la única información que contiene es el tipo de objeto y las coordenadas de la casilla que lo almacena.
- Sistema de representación visual para las casillas. Dependiendo del tipo de objeto almacenado en una casilla, se emplea una imagen u otra.
- Funciones para interactuar con las casillas, modificar su contenido y mostrar el contenido de la cuadrícula en la pantalla.

Sobre este proyecto se añaden las clases *Agent2D* y *CellRandomizer*. El agente en este caso incluye toda la lógica del entorno, que antes estaba distribuida entre el agente y los componentes *Pickable* y *GoalDetect* de los bloques:

- Inicialización del tablero vacío al comienzo de la ejecución.
- Sensores. La información que se recoge es el contenido de cada casilla – codificado mediante números enteros – y un booleano que indica si el agente está agarrando un bloque. Cuando se introduce el objetivo de construir un muro de más de un nivel se añade otro booleano que indica si se están completando los niveles en orden – es decir, si se termina un nivel antes de comenzar el siguiente.
- Acciones del agente. Movimiento en cuatro sentidos (incluye la rotación en el sentido del movimiento, igual que en los entornos anteriores), agarrar un bloque (de la casilla de enfrente) y soltar un bloque (en la casilla de enfrente).
- Detección de la colocación correcta de los bloques en la zona objetivo.
- Asignación de recompensas a la colocación de bloques y a la compleción de niveles. Las recompensas parciales son mayores si los niveles se construyen en orden.

La clase *CellRandomizer* contiene métodos que colocan de forma aleatoria los componentes del entorno en las casillas del tablero. S y se emplea una vez al comienzo de cada episodio.

Por último, se dibuja el *tileset* (ver *Figura 4.40*) – una textura 2D compuesta por todas las imágenes que representan a los objetos –. En la primera fila, de izquierda a derecha, se representan: casilla vacía (negro), bloque (amarillo), zona objetivo (azul), bloque colocado en el objetivo (verde) y dos bloques apilados en el objetivo (morado). En la segunda fila se representa el agente en sus diferentes orientaciones.

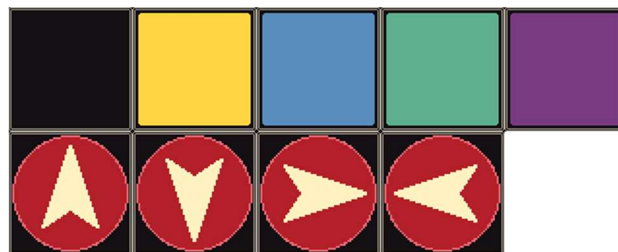

Figura 4.40: Tileset.

4.4.1. Versiones del entorno

La primera versión del entorno bidimensional (ver *Figura 4.41*) tuvo como objetivo la colocación de bloques dentro de la zona de construcción en un único nivel, no permitiendo el apilamiento de bloques. La acción de soltar un bloque está restringida a casillas vacías (negras) y casillas del objetivo (azules).

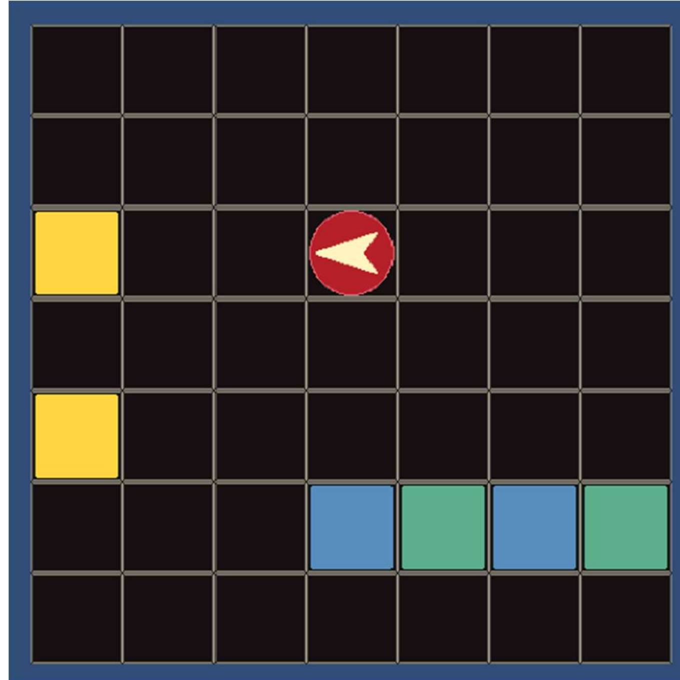

Figura 4.41: Primer entorno bidimensional.

La segunda versión del entorno (ver *Figura 4.42*) introduce la construcción en dos niveles, con el objetivo de enseñar al agente a seguir estrictamente el orden de construcción: no comenzar el segundo nivel hasta completar el primero. Se añade la posibilidad de soltar bloques sobre los bloques colocados (casillas verdes).

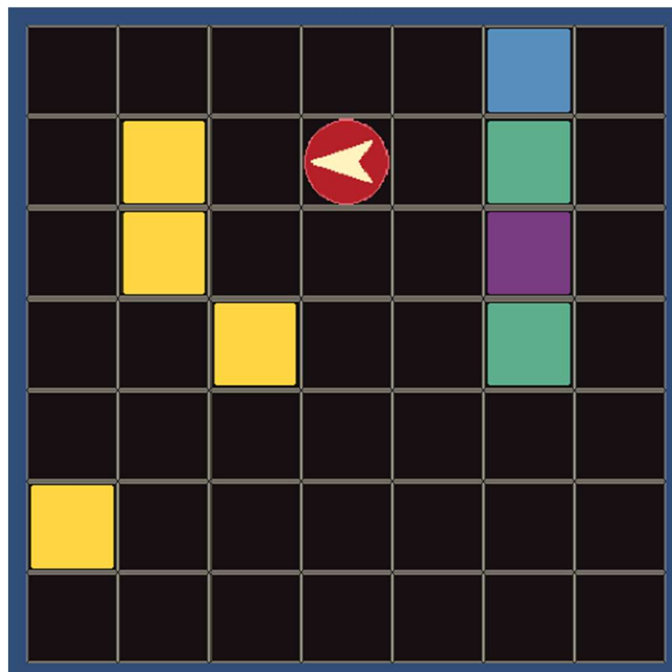

Figura 4.42: Segundo entorno bidimensional.

A continuación, se muestran una serie de figuras que resumen un episodio de ejemplo, en un entorno de tamaño 6x6, en el que el objetivo es completar dos niveles del muro. En este ejemplo, el agente realiza la tarea de forma óptima en seis pasos, maximizando la recompensa obtenida:

1. En el primer paso, se muestra el comienzo del episodio en el que hay seis bloques en el área y el objetivo está vacío. Ver *Figura 4.43*.
2. En el segundo paso, el agente se ha desplazado hasta un bloque y lo ha cogido. Ver *Figura 4.43*.
3. En el tercer paso, el agente ha colocado el bloque dentro del objetivo. Ver *Figura 4.44*.
4. El cuarto paso muestra el resultado de la misma secuencia de acciones con otros dos bloques, completando el primer nivel del muro. Ver *Figura 4.44*.
5. En el quinto paso, el agente ha cogido otro bloque y lo ha apilado sobre un bloque de los previamente colocados. Ver *Figura 4.45*.
6. En el sexto paso, el agente ha cogido el último bloque y se dispone a colocarlo, completando así el segundo nivel del muro y finalizando el episodio. Ver *Figura 4.45*.

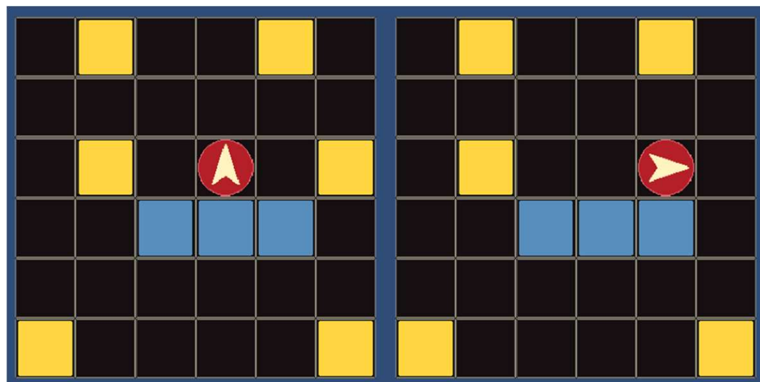

Figura 4.43: Primeros dos pasos del ejemplo.

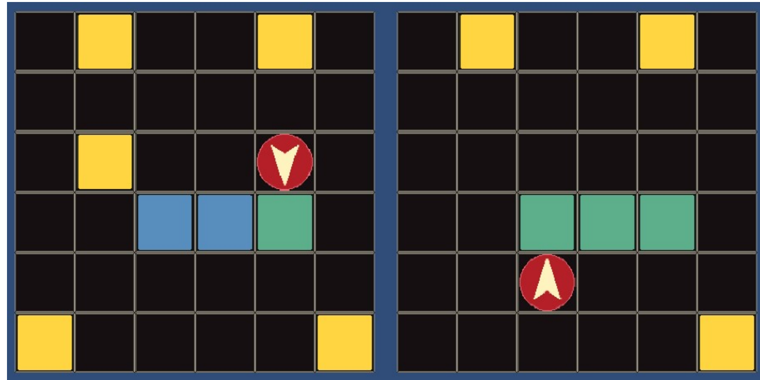

Figura 4.44: Pasos tercero y cuarto del ejemplo.

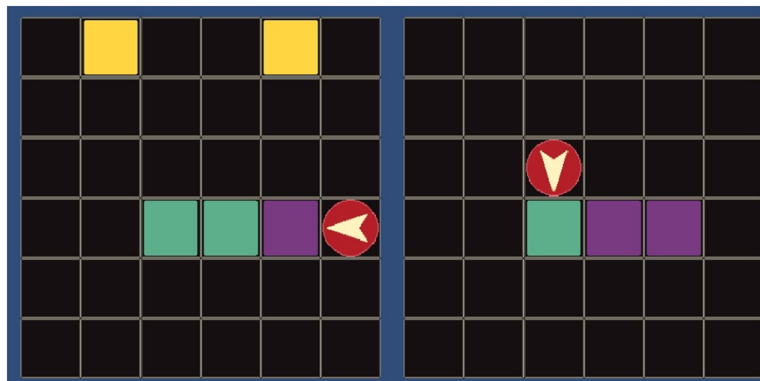

Figura 4.45: Pasos quinto y sexto del ejemplo.

Entrenamiento y resultados

El primer entrenamiento realizado fue con un entorno muy sencillo de 3x3 casillas (ver *Figura 4.46*), con el objetivo de colocar dos bloques, en el que lo que se buscaba realmente era encontrar una configuración de hiperparámetros que sirviera como base para los entrenamientos de este entorno. La única métrica empleada es la recompensa acumulada, ya que solo hay una tarea y fuente de recompensas.

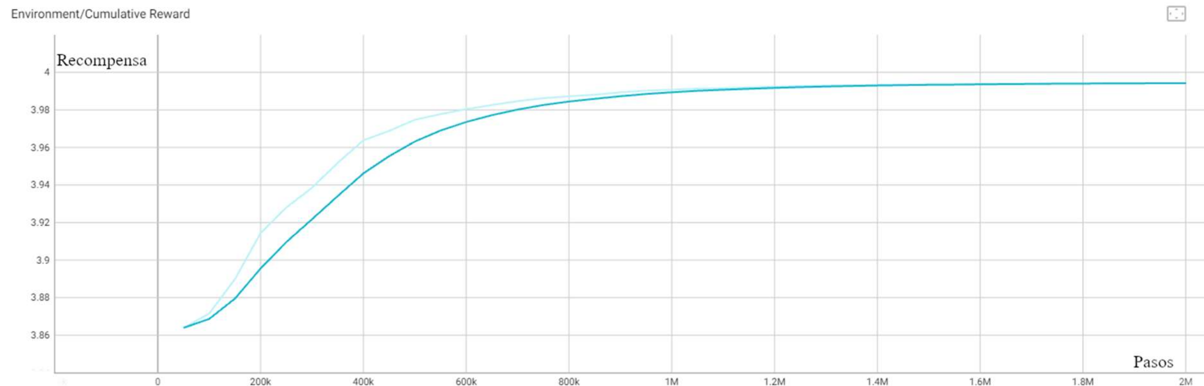

Figura 4.46: Primer entrenamiento. Entorno 3x3 con dos bloques.

Una vez completado satisfactoriamente el entrenamiento anterior, se realizan los primeros entrenamientos en un entorno algo más complejo, de 5x5 casillas y un objetivo de tres bloques. La dificultad de la tarea ha aumentado considerablemente, aunque el entorno sea aún bastante sencillo. Como puede verse en la *Figura 4.47*, el entrenamiento ha requerido cerca de ocho millones de pasos para estabilizarse.

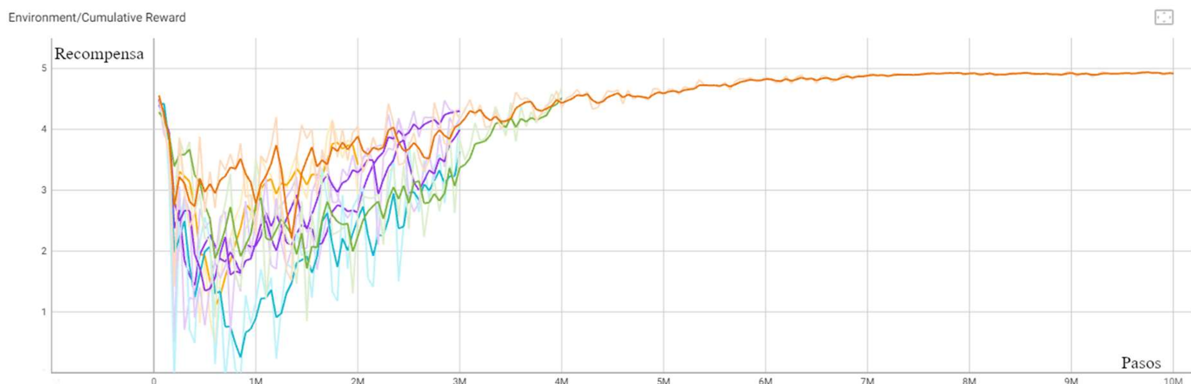

Figura 4.47: Entrenamientos de un entorno 5x5 con tres bloques.

A continuación, se emplea un entorno ligeramente más complejo, de 6x6 casillas, y manteniendo el objetivo de tres bloques. El tiempo necesario para el entrenamiento correcto aumenta de manera muy notable de nuevo, como se muestra en la *Figura 4.48*.

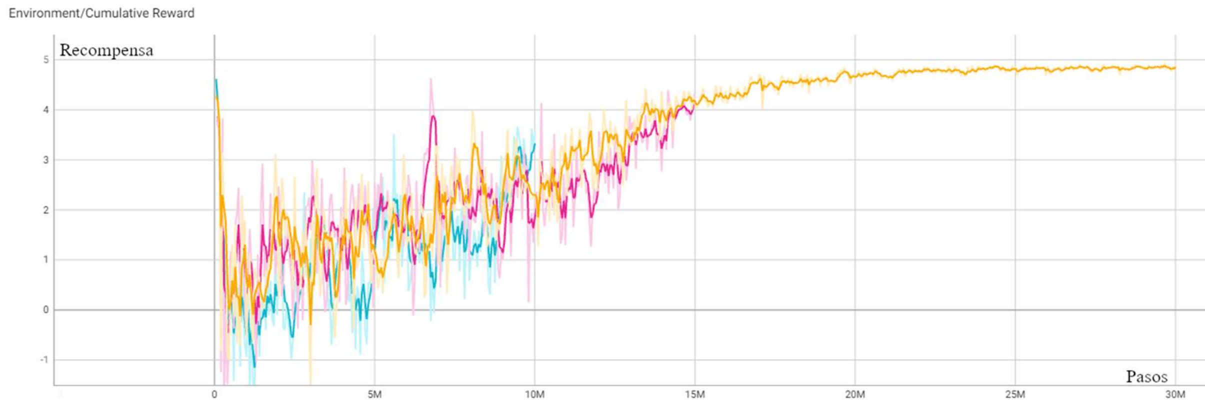

Figura 4.48: Entrenamiento del entorno 6x6 con 3 bloques.

En este momento, habiendo logrado enseñar a un agente la tarea de forma satisfactoria, se desarrolla la segunda versión del entorno, con dos niveles. Para analizar los entrenamientos, se añaden métricas adicionales:

- Pasos necesarios para colocar el bloque *n*, donde *n* = {1, 2, 3…}, contados a partir de la colocación del bloque *n-1* (o desde el comienzo del episodio para el primero). De los entornos anteriores se ha aprendido que encontrar un bloque y colocarlo es una tarea que se complica cuantos menos bloques queden disponibles, por lo que tiene sentido monitorizarlos de forma individual.
- Pasos necesarios para completar el primer nivel y pasos necesarios para completar el muro. Estas métricas ya fueron empleadas en entornos anteriores.
- Compleción del primer nivel en orden. Indica la fracción de episodios que completan el primer nivel antes de comenzar el segundo, que es la parte fundamental de la tarea a aprender.

El primer entorno entrenado fue de dimensiones 5x5, con seis bloques que deben colocarse en un objetivo de anchura 3. De nuevo, se prueban entrenamientos con un mayor número de pasos hasta obtener un resultado satisfactorio (ver *Figura 4.49*).

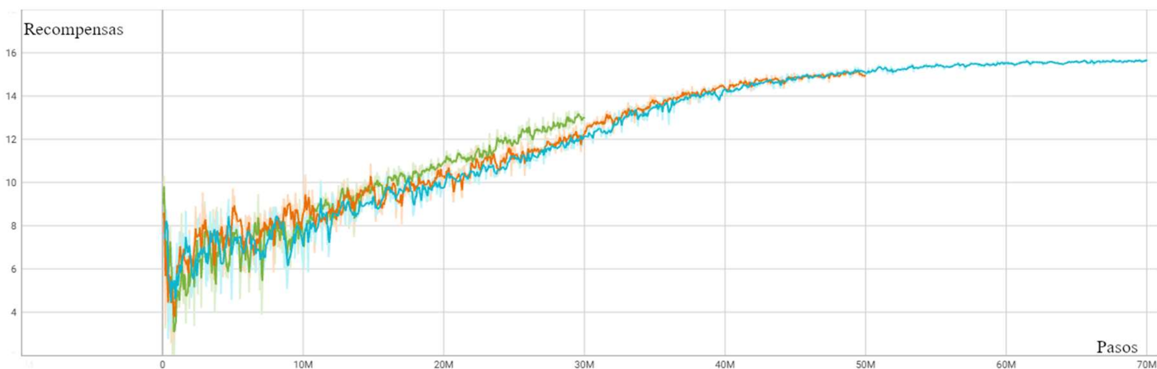

Figura 4.49: Recompensa acumulada. Entorno 5x5 con dos niveles y seis bloques.

La recompensa logra estabilizarse en un valor que tiende a 16, el máximo posible. Nótese cómo la longitud del entrenamiento es de 70 millones de pasos – y podría alargarse un poco más –, mientras que el entorno 5x5 con un solo nivel lograba la recompensa máxima tras unos 8 millones de pasos.

La gráfica ilustrada en la *Figura 4.50* muestra un rasgo importante del aprendizaje: se comienza a aprender desde el principio a colocar bloques, pero no se comienza a aprender a respetar el orden hasta cierto punto del entrenamiento – una vez completados unos 15 millones de pasos, aproximadamente. Comparando con la gráfica de recompensa acumulada (ver *Figura 4.49*), este punto coincide, más o menos, con el punto en el que la recompensa tiende a 10, que es el valor máximo que se obtiene al colocar todos los bloques sin respetar el orden de los niveles.

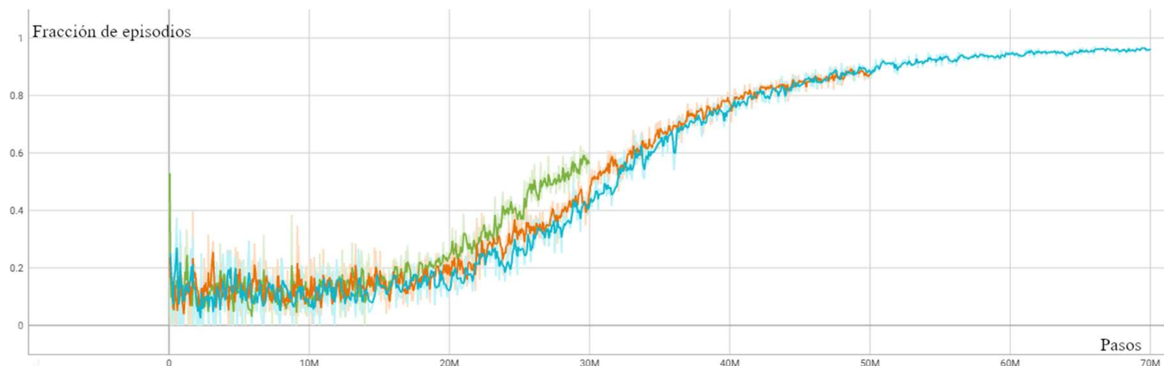

Figura 4.50: Fracción de episodios que completan el primer nivel en orden.

Este rasgo, observado también en entrenamientos posteriores, da a entender que es necesario cierto dominio de las tareas más simples antes de que el agente sea capaz de comenzar a aprender tareas compuestas. La eficiencia de las subtareas no tiene por qué ser muy buena para comenzar este aprendizaje, pero la velocidad de aprendizaje se incrementa

notablemente cuando se alcanza una gran eficiencia en las subtareas, como puede comprobarse en la gráfica de pasos necesarios para colocar el sexto y último bloque (ver *Figura 4.51*).

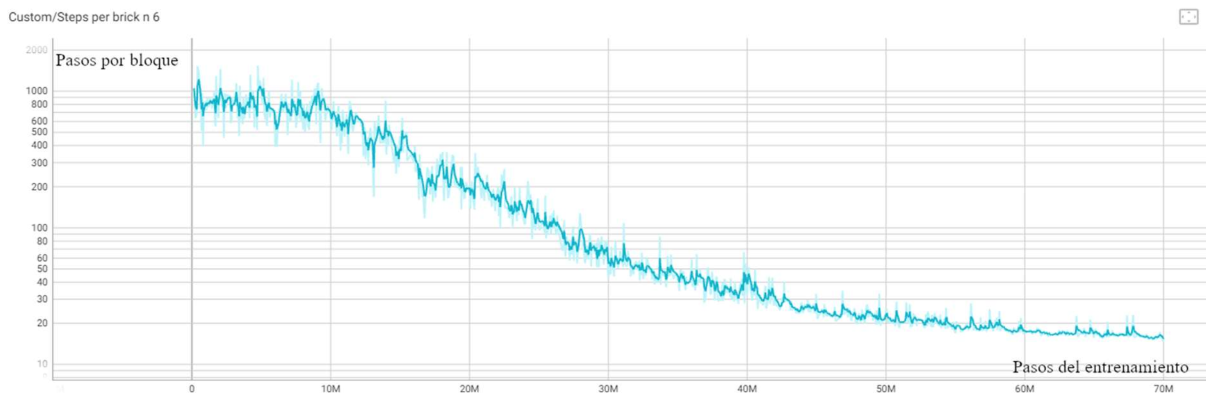

Figura 4.51: Pasos necesarios para colocar el sexto y último bloque.

A continuación, se pasa a un entorno de dimensiones 6x6 con ocho bloques que se colocan en un objetivo de cuatro casillas de anchura.

El primer entrenamiento no era capaz de aprender a colocar los bloques en orden. Con 100 millones de pasos de entrenamiento, el modelo entrenado era capaz de colocar todos los bloques, aunque su movimiento no fuera tan refinado como el obtenido en el entorno 5x5. En la distribución mostrada en la *Figura 4.52* puede verse que la gran mayoría de episodios se agrupan alrededor de una recompensa de 11,5, correspondiente a colocar los bloques de forma desordenada, y muy pocos obtienen una recompensa superior.

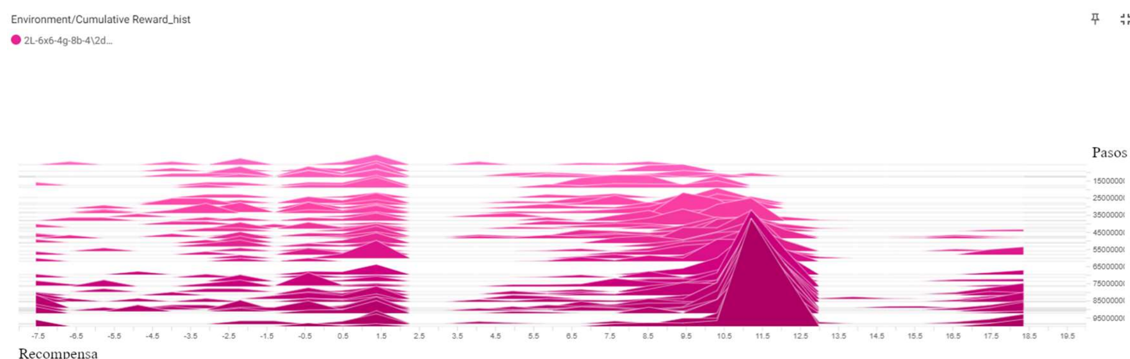

Figura 4.52: Distribución de recompensas en el primer entrenamiento del entorno 6x6.

Este problema está relacionado con el tamaño de la red neuronal empleada en el entrenamiento. Al aumentar el número de filas y columnas del entorno, el tamaño del vector de observación crece de forma cuadrática – y ese vector son los datos de entrada de la red.

La solución fue pasar de una red con tres capas ocultas de 256 neuronas a una red con cuatro capas ocultas de 512 neuronas.

El segundo entrenamiento, en naranja en la *Figura 4.53*, vuelve a ser capaz de aprender a construir los niveles en orden. Sin embargo, se vuelve a encontrar la necesidad de aumentar significativamente la longitud del entrenamiento. Se realizan unos últimos entrenamientos, llegando a durar el más largo 36 horas (ver *Figura 4.54*).

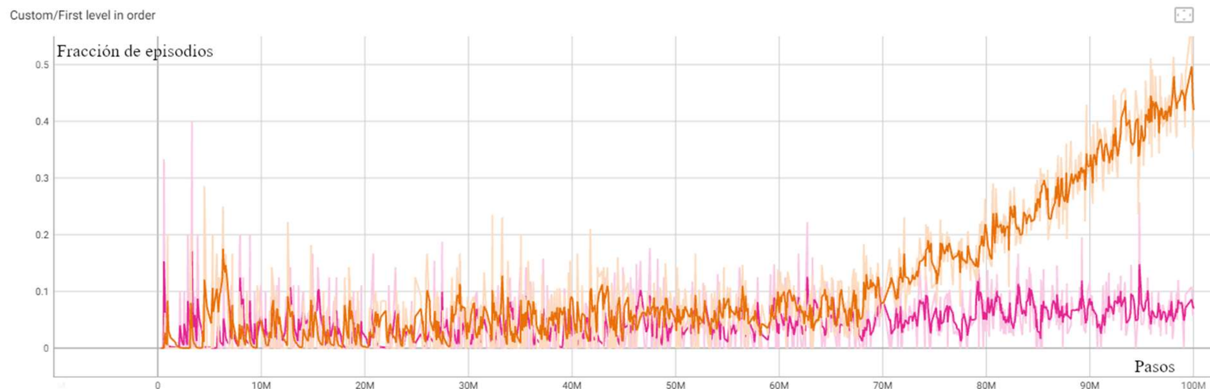

Figura 4.53: Aprendizaje de la construcción de dos niveles en orden.

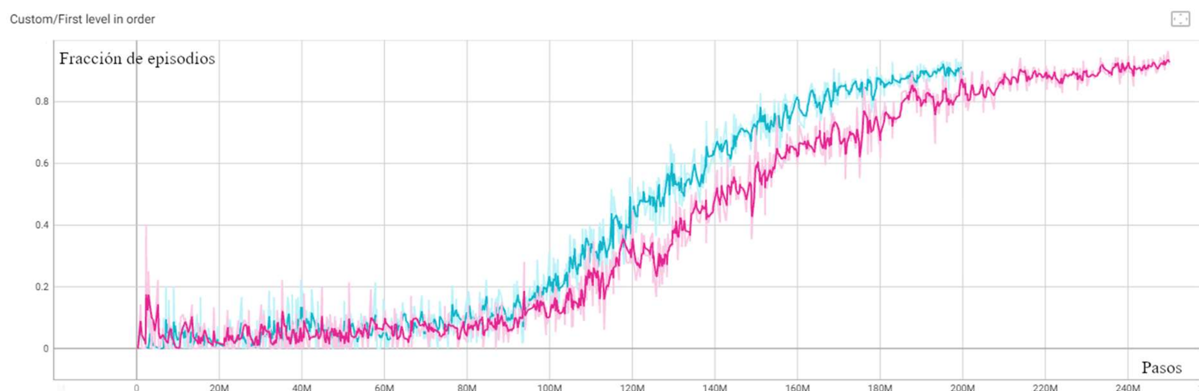

Figura 4.54: Últimos entrenamientos del entorno 6x6 con 8 bloques.

El resultado de ambos entrenamientos es satisfactorio, obteniendo un modelo entrenado que completa la tarea correctamente en la inmensa mayoría de episodios. Su eficiencia también es muy alta, moviéndose de forma directa entre los bloques que recoge y el objetivo.

4.5. SEPARACIÓN DE ENTORNOS

Por defecto, el comando Python que lanza el entrenamiento de un agente ML-Agents se comunica con Unity y entrena al agente en la escena abierta en el editor del programa. No obstante, también es posible utilizar un fichero ejecutable como entorno, que debe haber sido compilado con el editor de Unity previamente.

Esta opción ofrece dos ventajas: permite seguir trabajando con el editor mientras el entrenamiento se ejecuta en segundo plano y crea la posibilidad de desarrollar el entorno en una máquina y exportarlo a otras para el entrenamiento. Este apartado se centra este último caso.

El entrenamiento es un proceso que puede llegar a consumir muchos recursos, especialmente de GPU. Para no depender únicamente de la capacidad de procesamiento del ordenador en el que se desarrolla el entorno, se empleó un cuaderno Jupyter para entrenar en Google Colab.

El cuaderno es una modificación del empleado por Hugging Face en su curso *Deep RL Course* [40]. Incluye comandos Python para instalar ML-Agents, cargar el entorno de entrenamiento desde Google Drive – o se puede cargar manualmente–, configurar los hiperparámetros, lanzar el entrenamiento, monitorizarlo con TensorBoard y descargar los resultados.

El entrenamiento en Colab no fue empleado más allá del desarrollo inicial del cuaderno, ya que el tiempo de entrenamiento no disminuía respecto al entrenamiento local, que en aquel momento no era superior a las cuatro o cinco horas. El servicio gratuito de Colab desconecta el entorno de ejecución automáticamente tras alcanzar cierto tiempo de uso – limitado diariamente –, lo que no incentivó seguir explorando su viabilidad.

Es probable que el uso de un servicio de pago en la nube, con una mayor potencia de procesamiento GPU, hubiera sido útil para acortar los entrenamientos tan largos que se alcanzaron en las últimas versiones de los entornos 3D y 2D, resultando imprescindible en futuros desarrollos y ampliaciones del presente trabajo.y para cualquier línea de desarrollo futura.

Pese a no haber sido muy empleados, los cuadernos son perfectamente funcionales para ejecutar cualquier entrenamiento en un servicio en la nube o en un entorno Python en una máquina local. Los cambios necesarios para adaptar un cuaderno para un nuevo entorno Unity están detallados dentro del propio cuaderno, siendo una herramienta genérica y fácil de usar.

5. Conclusiones y trabajos futuros

El carácter de investigación del proyecto y las diferentes complicaciones imprevistas a lo largo de su desarrollo modificaron en cierta medida el objetivo inicial, cambiando en varias ocasiones la definición del muro que se pretendía construir. En última instancia, la compleción de un muro fue lograda en el entorno bidimensional. En el entorno tridimensional, sin embargo, únicamente se han conseguido buenos resultados en la subtarea de colocar bloques, sin lograr a alcanzar el objetivo final definido para ese entorno.

A continuación, se detallan las conclusiones de este Trabajo Fin de Grado junto con varias líneas de trabajo futuras, que permitirían seguir mejorando los modelos propuestos, especialmente el agente del entorno tridimensional, que bien por falta de tiempo o por salirse del alcance del proyecto no se siguieron durante el desarrollo.

5.1. CONCLUSIONES

El objetivo principal de este TFG era desarrollar un entorno en Unity en el que un agente entrenado mediante técnicas de *reinforcement learning* fuese capaz de construir un muro. Tal y como se menciona al inicio de este capítulo, el objetivo se ha alcanzado en un entorno 2D mientras que en el entorno 3D se han conseguido buenos resultados en la tarea de colocar bloques.

A continuación, se detallan las conclusiones alcanzadas en el desarrollo de este proyecto:

- Es posible entrenar un agente con Unity y ML-Agents que coloque bloques de forma ordenada, aplicable a la construcción de muros. El último agente entrenado en un entorno 2D supera el 90% de fiabilidad en una tarea compleja.
- En los entornos tridimensionales, los entrenamientos son más complejos por lo que requieren más pasos para que el agente sea capaz de aprender a realizar la tarea principal. Los agentes en el último entorno 3D son fiables en la realización de la subtarea de colocar todos los bloques, pero no fueron capaces de aprender la tarea más compleja – algo posible, como se demuestra en los entornos más sencillos. Algunas soluciones a este problema son el empleo una mayor capacidad de cómputo y otras técnicas que se detallan en la Sección 5.2.
- El planteamiento del entrenamiento de los agentes ha sido correcto, tanto el sistema de recompensas diseñado como la selección de hiperparámetros en los entrenamientos exitosos de cada entorno.

Desde el punto de vista personal, la realización de este proyecto ha representado un avance significativo en mi formación, dado que me ha permitido adquirir conocimientos en el campo del aprendizaje por refuerzo y explotar la plataforma Unity.

5.2. TRABAJO FUTURO

A continuación, se presenta una serie de líneas de trabajo que permitirían mejorar los modelos a partir de su estado actual:

1. Acercar el diseño de los sensores del entorno 3D a los sensores del entorno 2D. Este cambio requiere desarrollar los sensores de forma manual en vez de emplear componentes como los *Ray Perception Sensor* empleados. Por ejemplo, unos sensores que dividan el área observable en casillas y detecten si hay un bloque en cada una o no, aportando un valor booleano por cada casilla, algo mucho más sencillo que los datos de las colisiones detectadas por los sensores tipo *raycast*. Un sistema de sensores de este estilo equivale a una cámara que observa el espacio desde una posición cenital y se comunica con el agente, lo que sigue siendo realista.

2. Otro planteamiento, que se puede combinar con el anterior, consistiría en dividir la tarea de colocar un bloque en dos problemas más sencillos. La primera tarea se limitaría a encontrar los bloques y llevarlos hasta el objetivo, algo en lo que ya se ha conseguido una buena eficiencia. La segunda tarea se centraría en colocar el bloque en la posición deseada – la posición libre con la mayor recompensa. Estas dos subtareas tienen, cada una, menos estados y acciones que el problema inicial y pueden ser entrenadas de forma independiente, simplificando notablemente el proceso de entrenamiento. Este enfoque requiere, sin embargo, el desarrollo de tres entornos: uno para cada subtarea y uno en el que el agente integre ambos modelos y gestione la petición de decisiones a cada uno de ellos.

3. El entorno bidimensional puede emplearse para implementar nuevas tareas complejas antes de intentar aplicarlas a un entorno tridimensional. Por ejemplo, se puede integrar la construcción según el aparejo de soga, tarea que el agente debería ser capaz de aprender en el entorno 2D, al igual que la construcción por niveles. El objetivo es asegurarse rápidamente de tener un buen sistema de recompensas que permita su entrenamiento adecuado, y trasladarlo a un entorno tridimensional mediante técnicas como la propuesta en el primer punto de este apartado.

6. Bibliografía

[1] IBM. What is Artificial Intelligence (AI)? Accedido por última vez: 10 de julio de 2023. https://www.ibm.com/topics/artificial-intelligence

[2] González Valenzuela, C. (27 de mayo de 2023) ¿Qué es el 'Deep Learning' y por qué se considera una revolución en la inteligencia artificial? Accedido por última vez: 10 de julio de 2023. https://computerhoy.com/tecnologia/deep-learning-considera-revolucion-ia-1241180

[3] IBM. What is Machine Learning? Accedido por última vez: 10 de julio de 2023. https://www.ibm.com/topics/machine-learning

[4] Sutton, R. S. y Barto, A. G. (2018). *Reinforcement Learning: An Introduction (2ª ed.)* The MIT Press.

[5] Bowyer, C. M. A Crude History of Reinforcement Learning (RL). Accedido por última vez: 10 de julio de 2023. https://medium.com/@CalebMBowyer/a-crude-history-of-reinforcement-learning-rl-1abaae72550e

[6] Doggers, P. (18 noviembre de 2021). How AlphaZero Learns Chess. Accedido por última vez: 10 de julio de 2023. https://www.chess.com/news/view/how-alphazero-learns-chess

[7] Schulman, J., Klimov, O., Wolski, F., Dhariwal, P. y Radford, A. (20 de julio de 2017). Proximal Policy Optimization. Accedido por última vez: 10 de julio de 2023. https://openai.com/research/openai-baselines-ppo

[8] OpenAI (2018). Proximal Policy Optimization – Spinning Up documentation. Accedido por última vez: 10 de julio de 2023. https://spinningup.openai.com/en/latest/algorithms/ppo.html

[9] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T. & Silver, D. (2020). Mastering atari, go, chess and shogi by planning with a learned model. *Nature, 588*(7839), 604-609.

[10] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science, 362(6419), 1140-1144.

[11] Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research, 47, 253-279.

[12] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*

[13] E. Marchesini and A. Farinelli, "Discrete Deep Reinforcement Learning for Mapless Navigation," 2020 IEEE International Conference on Robotics and Automation (ICRA), Paris, France, 2020, pp. 10688-10694, doi: 10.1109/ICRA40945.2020.9196739.

[14] Z. Cao and C. -T. Lin, "Reinforcement Learning From Hierarchical Critics," in IEEE Transactions on Neural Networks and Learning Systems, vol. 34, no. 2, pp. 1066-1073, Feb. 2023, doi: 10.1109/TNNLS.2021.3103642.

[15] Ward, T., Bolt, A., Hemmings, N., Carter, S., Sanchez, M., Barreira, R., Noury, S., Anderson, K., Lemmon, J., Coe, J., Trochim, P., Handley, T. & Bolton, A. (2020). Using unity to help solve intelligence. *arXiv preprint arXiv:2011.09294*

[16] Google Deepmind. The Challenge Match. Accedido por última vez: 10 de julio de 2023. https://www.deepmind.com/research/highlighted-research/alphago/the-challenge-match

[17] Unity Technologies. Unity – Manual: Physics. Accedido por última vez: 10 de julio de 2023. https://docs.unity3d.com/Manual/PhysicsSection.html

[18] Unity Technologies. Build More Engaging Games With ML Agents. Accedido por última vez: 10 de julio de 2023.https://unity.com/products/machine-learning-agents

[19] Welcome to Python.org. Accedido por última vez: 10 de julio de 2023. https://www.python.org

[20] Astropy. Accedido por última vez: 11 de julio de 2023. https://www.astropy.org

[21] Biopython. Accedido por última vez: 11 de julio de 2023. https://biopython.org

[2 Pyspark package. Accedido por última vez: 11 de julio de 2023. https://spark.apache.org/docs/2.1.0/api/python/pyspark.html

[23] Scikit-learn: machine learning in Python. Accedido por última vez: 11 de julio de 2023. https://scikit-learn.org/stable/

[24] Features | PyTorch. Accedido por última vez: 10 de julio de 2023. https://pytorch.org/features/

[25] TensorFlow. Accedido por última vez: 10 de julio de 2023. https://www.tensorflow.org

[26] Hoong, K (10 de febrero de 2023). How to use TensorBoard with PyTorch. Accedido por última vez: 10 de julio de 2023. https://kuanhoong.medium.com/how-to-use-tensorboard-with-pytorch-e2b84aa55e67

[27] Using TensorBoard to Observe Training – Unity ML-Agents Toolkit. Accedido por última vez: 10 de julio de 2023. https://unity-technologies.github.io/ml-agents/Using-Tensorboard/

[28] Unity Technologies. Unity – Manual: Creating and Using Scripts. Accedido por última vez: 10 de julio de 2023. https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html

[29] What is Git. | Atlassian Git Tutorial. Accedido por última vez: 10 de julio de 2023. https://www.atlassian.com/git/tutorials/what-is-git

[30] ¿Qué es GitLab? Accedido por última vez: 10 de julio de 2023. https://turingears.com/que-es-gitlab/

[31] Game Maker's Toolkit (2 de diciembre de 2022). The Unity Tutorial For Complete Beginners. Accedido por última vez: 10 de julio de 2023. https://www.youtube.com/watch?v=XtQMytORBmM

[32] Brackeys – YouTube. Accedido por última vez: 10 de julio de 2023. https://www.youtube.com/@Brackeys

[33] Comunidad de desarrolladores de ML-Agents (Última modificación: 15 de diciembre de 2021). Getting started at release 19. Accedido por última vez: 10 de julio de 2023. https://github.com/Unity-Technologies/ml-agents/blob/release_19_docs/docs/Getting-Started.md

[34] Comunidad de desarrolladores de ML-Agents (Última modificación: 1 de febrero de 2019). Training ML-Agents. Accedido por última vez: 10 de julio de 2023. https://github.com/bascoul/Ml-Agents/blob/master/docs/Training-ML-Agents.md

[35] Comunidad de desarrolladores de ML-Agents (Última modificación: 18 de noviembre de 2022). Training Configuration File. Accedido por última vez: 10 de julio de 2023. https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md

[36] What Is The Difference Between Deep Learning And Machine Learning? Quantdare. Accedido por última vez: 13 de julio de 2023. https://www.chegos.pl/what-is-the-difference-between-deep-learning-and-machine-pp-47006699

[37] Tipos de Aparejos – Construmática. Accedido por última vez: 10 de julio de 2023. https://www.construmatica.com/construpedia/Tipos_de_Aparejos

[38] Code Monkey (Última publicación: 5 de octubre de 2022). Grid System in Unity (How to make it and where to use it). Accedido por última vez: 10 de julio de 2023. https://www.youtube.com/playlist?list=PLzDRvYVwl53uhO8yhqxcyjDImRjO9W722

[39] Code Monkey (4 de octubre de 2019). Grid System in Unity (Heatmap, Pathfinding, Building Area). Accedido por última vez: 10 de julio de 2023. https://unitycodemonkey.com/video.php?v=waEsGu--9P8

[40] Simonini, T. Let’s train our agents – Hugging Face Deep RL Course. Accedido por última vez: 10 de julio de 2023. https://huggingface.co/learn/deep-rl-course/unit5/hands-on